Kostenloses eBook

LERNEN C++

Free unaffiliated eBook created from
Stack Overflow contributors.

#c++

Inhaltsverzeichnis

Über

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: cplusplus

It is an unofficial and free C++ ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official C++.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Kapitel 1: Erste Schritte mit C ++

Bemerkungen

Das 'Hello World'-Programm ist ein allgemeines Beispiel, das einfach zur Überprüfung der Anwesenheit von Compiler und Bibliothek verwendet werden kann. Es verwendet die C ++ - Standardbibliothek mit `std::cout` von `<iostream>` und hat nur eine zu kompilierende Datei, wodurch die Möglichkeit möglicher Benutzerfehler während der Kompilierung minimiert wird.

Das Kompilieren eines C ++ - Programms unterscheidet sich grundsätzlich zwischen Compilern und Betriebssystemen. Das Thema Kompilieren und Erstellen enthält Details zum Kompilieren von C ++ - Code auf verschiedenen Plattformen für verschiedene Compiler.

Versionen

Ausführung	Standard	Veröffentlichungsdatum
C ++ 98	ISO / IEC 14882: 1998	1998-09-01
C ++ 03	ISO / IEC 14882: 2003	2003-10-16
C ++ 11	ISO / IEC 14882: 2011	2011-09-01
C ++ 14	ISO / IEC 14882: 2014	2014-12-15
C ++ 17	TBD	2017-01-01
C ++ 20	TBD	2020-01-01

Examples

Hallo Welt

Dieses Programm druckt `Hello World!` zum Standardausgabestrom:

```
#include <iostream>

int main()
{
    std::cout << "Hello World!" << std::endl;
}
```

Sehen Sie live auf Coliru .

Analyse

Lassen Sie uns jeden Teil dieses Codes detailliert untersuchen:

- `#include <iostream>` ist eine **Präprozessoranweisung** , die den Inhalt der Standard-C ++ - Headerdatei `iostream` .

  `iostream` ist eine **Standard-Bibliothekskopfdatei** , die Definitionen der Standard-Eingabe- und Ausgabeströme enthält. Diese Definitionen sind im `std` Namespace enthalten, der unten erläutert wird.

  Die **Standard-E / A-Ströme** bieten Programmen die Möglichkeit, Eingaben von einem externen System zu erhalten und an dieses auszugeben - normalerweise das Terminal.

- `int main() { ... }` definiert eine neue Funktion namens `main` . Konventionell wird die `main` bei Ausführung des Programms aufgerufen. In einem C ++ - Programm darf nur eine `main` vorhanden sein, und es muss immer eine Zahl vom Typ `int` werden.

  Das `int` ist der Rückgabetyp der Funktion. Der von der `main` zurückgegebene Wert ist ein **Beendigungscode.**

  Konventionell wird ein Programm-Exit-Code von `0` oder `EXIT_SUCCESS` von einem System, das das Programm ausführt, als Erfolg interpretiert. Jeder andere Rückkehrcode ist mit einem Fehler verbunden.

  Wenn keine `return` - Anweisung vorhanden ist, die `main` (und damit das Programm selbst) liefert `0` standardmäßig. In diesem Beispiel müssen wir `return 0;` nicht explizit schreiben `return 0;` .

  Alle anderen Funktionen, mit Ausnahme derjenigen, die den `void` Typ zurückgeben, müssen explizit einen Wert entsprechend ihrem Rückgabetyp zurückgeben, sonst dürfen sie überhaupt nicht zurückgeben.

- `std::cout << "Hello World!" << std::endl;` druckt "Hallo Welt!" zum Standardausgabestrom:

  - `std` ist ein Namespace , und `::` ist der **Operator** für die **Bereichsauflösung** , der Lookups für Objekte nach Namen in einem Namespace ermöglicht.

    Es gibt viele Namespaces. Hier verwenden wir `::` zu zeigen, dass wir `cout` aus dem `std` Namespace verwenden möchten. Weitere Informationen finden Sie unter Scope Resolution Operator - Microsoft-Dokumentation .

  - `std::cout` ist die **Standardausgabe** Objekt, in definierten `iostream` , und druckt es auf die Standardausgabe (`stdout`).

  - `<<` ist *in diesem Zusammenhang* der **Stream-Einfügungsoperator** , der so genannt wird, weil er ein Objekt in das *Stream*- Objekt *einfügt* .

    Die Standardbibliothek definiert den Operator `<<` , um Daten für bestimmte Datentypen

in Ausgabeströme einzufügen. `stream << content` fügt `content` in den Stream ein und gibt denselben, aber aktualisierten Stream zurück. Dadurch können Stream-Einfügungen verkettet werden: `std::cout << "Foo" << " Bar";` druckt "FooBar" auf die Konsole.

- `"Hello World!"` ist ein **Zeichenkettenliteral** oder ein "Textliteral". Der Stream-Einfügeoperator für Zeichenkettenliterale wird in der Datei `iostream` definiert.

- `std::endl` ist ein spezielles **E / A-Stream-Manipulatorobjekt** , das auch in der Datei `iostream` definiert ist. Durch Einfügen eines Manipulators in einen Stream wird der Status des Streams geändert.

  Der Stream-Manipulator `std::endl` führt zwei Dinge aus: Zuerst fügt er das Zeilenende-Zeichen ein und leert dann den Stream-Puffer, damit der Text auf der Konsole angezeigt wird. Dadurch wird sichergestellt, dass die in den Stream eingefügten Daten tatsächlich auf Ihrer Konsole angezeigt werden. (Stream-Daten werden normalerweise in einem Puffer gespeichert und dann in Batches "geleert", sofern Sie nicht sofort eine Spülung erzwingen.)

  Eine alternative Methode, die den Flush vermeidet, ist:

  ```
  std::cout << "Hello World!\n";
  ```

  Dabei ist `\n` die **Escape-Zeichenfolge** für das Zeilenvorschubzeichen.

- Das Semikolon (`;`) benachrichtigt den Compiler darüber, dass eine Anweisung beendet wurde. Alle C ++ - Anweisungen und Klassendefinitionen erfordern ein abschließendes Semikolon.

Bemerkungen

Ein **Kommentar** ist eine Möglichkeit, beliebigen Text in den Quellcode einzufügen, ohne dass der C ++ - Compiler ihn mit funktionaler Bedeutung interpretiert. Kommentare werden verwendet, um Einblick in das Design oder die Methode eines Programms zu geben.

Es gibt zwei Arten von Kommentaren in C ++:

Einzeilige Kommentare

Die doppelte Schrägstrichfolge `//` markiert den gesamten Text bis zu einer neuen Zeile als Kommentar:

```
int main()
{
   // This is a single-line comment.
   int a;  // this also is a single-line comment
   int i;  // this is another single-line comment
}
```

C-Style / Block-Kommentare

Mit der Sequenz `/*` wird der Beginn des Kommentarblocks und mit der Sequenz `*/` das Ende des Kommentars angegeben. Der gesamte Text zwischen den Start- und Endsequenzen wird als Kommentar interpretiert, auch wenn der Text ansonsten eine gültige C ++ - Syntax ist. Diese werden manchmal als "C-style" -Kommentare bezeichnet, da diese Kommentarsyntax von der C ++ - Vorgängersprache C übernommen wird:

```
int main()
{
   /*
    *  This is a block comment.
    */
   int a;
}
```

In jedem Blockkommentar können Sie alles schreiben, was Sie möchten. Wenn der Compiler auf das Symbol `*/` stößt, wird der Blockkommentar abgebrochen:

```
int main()
{
   /* A block comment with the symbol /*
      Note that the compiler is not affected by the second /*
      however, once the end-block-comment symbol is reached,
      the comment ends.
   */
   int a;
}
```

Das obige Beispiel ist gültiger C ++ - Code (und C-Code). Wenn Sie jedoch `/*` in einem Blockkommentar enthalten, kann dies bei einigen Compilern zu einer Warnung führen.

Block-Kommentare können auch *innerhalb* einer einzelnen Zeile beginnen und enden. Zum Beispiel:

```
void SomeFunction(/* argument 1 */ int a, /* argument 2 */ int b);
```

Bedeutung der Kommentare

Wie bei allen Programmiersprachen bieten Kommentare mehrere Vorteile:

- Explizite Dokumentation des Codes, um das Lesen / Verwalten zu erleichtern
- Erläuterung des Zweckes und der Funktionalität des Codes
- Details zum Verlauf oder zur Begründung des Codes
- Platzierung von Copyright / Lizenzen, Projektnotizen, besonderen Dank, Mitwirkenden, etc. direkt im Quellcode.

Kommentare haben jedoch auch ihre Nachteile:

- Sie müssen beibehalten werden, um Änderungen im Code widerzuspiegeln
- Exzessive Kommentare machen den Code *weniger* lesbar

Der Bedarf an Kommentaren kann reduziert werden, indem klarer, selbstdokumentierender Code geschrieben wird. Ein einfaches Beispiel ist die Verwendung von erklärenden Namen für Variablen, Funktionen und Typen. Das Ausrechnen von logisch zusammenhängenden Aufgaben in diskrete Funktionen geht damit Hand in Hand.

Kommentarmarken zum Deaktivieren von Code

Während der Entwicklung können Kommentare auch dazu verwendet werden, Teile des Codes schnell zu deaktivieren, ohne ihn zu löschen. Dies ist häufig für Test- oder Debugging-Zwecke hilfreich, eignet sich jedoch nur für temporäre Bearbeitungen. Dies wird oft als "Auskommentieren" bezeichnet.

In ähnlicher Weise ist es verpönt, alte Versionen eines Codes in einem Kommentar zu Referenzzwecken zu behalten, da dies Dateien stört und dabei wenig Wert bietet, verglichen mit der Erforschung der Geschichte des Codes über ein Versionssystem.

Funktion

Eine **Funktion** ist eine Codeeinheit, die eine Folge von Anweisungen darstellt.

Funktionen können **Argumente** oder Werte annehmen und einen einzelnen Wert **zurückgeben** (oder nicht). Um eine Funktion zu verwenden, wird ein **Funktionsaufruf** für Argumentwerte verwendet und die Verwendung des Funktionsaufrufs selbst wird durch seinen Rückgabewert ersetzt.

Jede Funktion verfügt über eine **Typensignatur** - die Typen ihrer Argumente und den Typ ihres Rückgabetyps.

Funktionen werden von den Konzepten der Prozedur und der mathematischen Funktion inspiriert.

- Hinweis: C ++ - Funktionen sind im Wesentlichen Prozeduren und folgen nicht der genauen Definition oder den Regeln mathematischer Funktionen.

Funktionen dienen häufig dazu, eine bestimmte Aufgabe auszuführen. und kann von anderen Teilen eines Programms aufgerufen werden. Eine Funktion muss deklariert und definiert werden, bevor sie an anderer Stelle in einem Programm aufgerufen wird.

- Hinweis: Beliebte Funktionsdefinitionen können in anderen enthaltenen Dateien versteckt sein (häufig zur Vereinfachung und Wiederverwendung in vielen Dateien). Dies ist eine übliche Verwendung von Header-Dateien.

Funktionserklärung

Eine **Funktionsdeklaration** gibt dem Compiler die Existenz einer Funktion mit ihrem Namen und ihrer Typensignatur an. Die Syntax lautet wie folgt:

```
int add2(int i); // The function is of the type (int) -> (int)
```

Im obigen Beispiel erklärt die Funktion `int add2(int i)` dem Compiler Folgendes:

- Der **Rückgabetyp** ist `int` .
- Der **Name** der Funktion lautet `add2` .
- Die **Anzahl der Argumente** für die Funktion ist 1:
    - Das erste Argument ist vom Typ `int` .
    - Das erste Argument wird im Inhalt der Funktion mit dem Namen `i` .

Der Argumentname ist optional. Die Deklaration für die Funktion könnte auch folgende sein:

```
int add2(int); // Omitting the function arguments' name is also permitted.
```

Gemäß der **Eindefinitionsregel** kann eine Funktion mit einer bestimmten Typensignatur nur einmal in einer gesamten C ++ - Codebasis deklariert oder definiert werden, die für den C ++ - Compiler sichtbar ist. Das heißt, Funktionen mit einer bestimmten Typensignatur können nicht neu definiert werden - sie müssen nur einmal definiert werden. Folgendes ist daher kein gültiges C ++:

```
int add2(int i);  // The compiler will note that add2 is a function (int) -> int
int add2(int j);  // As add2 already has a definition of (int) -> int, the compiler
                  // will regard this as an error.
```

Wenn eine Funktion nichts zurückgibt, wird ihr Rückgabetyp als `void` . Wenn keine Parameter erforderlich sind, sollte die Parameterliste leer sein.

```
void do_something(); // The function takes no parameters, and does not return anything.
                     // Note that it can still affect variables it has access to.
```

Funktionsaufruf

Eine Funktion kann aufgerufen werden, nachdem sie deklariert wurde. Das folgende Programm ruft beispielsweise `add2` mit dem Wert `2` innerhalb der Funktion von `main` :

```
#include <iostream>

int add2(int i);    // Declaration of add2

// Note: add2 is still missing a DEFINITION.
// Even though it doesn't appear directly in code,
// add2's definition may be LINKED in from another object file.
```

```
int main()
{
    std::cout << add2(2) << "\n";  // add2(2) will be evaluated at this point,
                                   // and the result is printed.
    return 0;
}
```

Hier ist `add2(2)` die Syntax für einen Funktionsaufruf.

Funktionsdefinition

Eine *Funktionsdefinition* * ähnelt einer Deklaration, enthält jedoch auch den Code, der ausgeführt wird, wenn die Funktion innerhalb ihres Rumpfes aufgerufen wird.

Ein Beispiel für eine Funktionsdefinition für `add2` könnte sein:

```
int add2(int i)       // Data that is passed into (int i) will be referred to by the name i
{                     // while in the function's curly brackets or "scope."

    int j = i + 2;    // Definition of a variable j as the value of i+2.
    return j;         // Returning or, in essence, substitution of j for a function call to
                      // add2.
}
```

Funktionsüberladung

Sie können mehrere Funktionen mit demselben Namen, aber unterschiedlichen Parametern erstellen.

```
int add2(int i)           // Code contained in this definition will be evaluated
{                         // when add2() is called with one parameter.
    int j = i + 2;
    return j;
}

int add2(int i, int j)    // However, when add2() is called with two parameters, the
{                         // code from the initial declaration will be overloaded,
    int k = i + j + 2 ;   // and the code in this declaration will be evaluated
    return k;             // instead.
}
```

Beide Funktionen werden mit dem gleichen Namen `add2` . Die eigentliche Funktion, die aufgerufen wird, hängt jedoch direkt von der Menge und dem Typ der Parameter im Aufruf ab. In den meisten Fällen kann der C ++ - Compiler berechnen, welche Funktion aufgerufen werden soll. In einigen Fällen muss der Typ explizit angegeben werden.

Standardparameter

Standardwerte für Funktionsparameter können nur in Funktionsdeklarationen angegeben werden.

```
int multiply(int a, int b = 7); // b has default value of 7.
int multiply(int a, int b)
{
    return a * b;               // If multiply() is called with one parameter, the
}                               // value will be multiplied by the default, 7.
```

In diesem Beispiel kann `multiply()` mit einem oder zwei Parametern aufgerufen werden. Wenn nur ein Parameter angegeben wird, hat `b` den Standardwert 7. Standardargumente müssen in den letzten Argumenten der Funktion angegeben werden. Zum Beispiel:

```
int multiply(int a = 10, int b = 20); // This is legal
int multiply(int a = 10, int b);      // This is illegal since int a is in the former
```

Spezielle Funktionsaufrufe - Operatoren

Es gibt spezielle Funktionsaufrufe in C ++, die eine andere Syntax als `name_of_function(value1, value2, value3)` . Das häufigste Beispiel sind Operatoren.

Bestimmte spezielle Zeichenfolgen, die vom Compiler auf Funktionsaufrufe reduziert werden, z. B. `!` , `+` , `-` , `*` , `%` und `<<` und viele mehr. Diese Sonderzeichen stehen normalerweise im Zusammenhang mit der nicht-programmierenden Verwendung oder werden für Ästhetik verwendet (z. B. wird das Zeichen `+` sowohl in der C ++ - Programmierung als auch in der Mathematik als Zusatzsymbol erkannt).

C ++ behandelt diese Zeichenfolgen mit einer speziellen Syntax. Im Wesentlichen wird jedoch jedes Auftreten eines Operators auf einen Funktionsaufruf reduziert. Beispielsweise der folgende C ++ - Ausdruck:

```
3+3
```

entspricht dem folgenden Funktionsaufruf:

```
operator+(3, 3)
```

Alle Namen der `operator` beginnen mit dem `operator` .

Während in C ++ unmittelbarem Vorgänger C die Namen der Operatorfunktionen nicht durch Angabe zusätzlicher Definitionen mit unterschiedlichen Typensignaturen unterschiedlichen Bedeutungen zugewiesen werden können, ist dies in C ++ gültig. Das "Ausblenden" zusätzlicher Funktionsdefinitionen unter einem eindeutigen Funktionsnamen wird in C ++ als **Operatorüberladung bezeichnet** und ist in C ++ eine relativ verbreitete, aber keine universelle Konvention.

Sichtbarkeit von Funktionsprototypen und Deklarationen

In C ++ muss Code vor der Verwendung deklariert oder definiert werden. Das folgende Beispiel erzeugt beispielsweise einen Fehler bei der Kompilierung:

```
int main()
{
  foo(2); // error: foo is called, but has not yet been declared
}

void foo(int x) // this later definition is not known in main
{
}
```

Es gibt zwei Möglichkeiten, dies zu beheben: Setzen Sie entweder die Definition oder Deklaration von `foo()` vor seiner Verwendung in `main()` . Hier ist ein Beispiel:

```
void foo(int x) {}  //Declare the foo function and body first

int main()
{
  foo(2); // OK: foo is completely defined beforehand, so it can be called here.
}
```

Es ist jedoch auch möglich, die Funktion "vorwärts zu deklarieren", indem nur eine "Prototyp" - Deklaration vor ihrer Verwendung gesetzt wird und der Funktionskörper später definiert wird:

```
void foo(int);  // Prototype declaration of foo, seen by main
                // Must specify return type, name, and argument list types
int main()
{
  foo(2); // OK: foo is known, called even though its body is not yet defined
}

void foo(int x) //Must match the prototype
{
    // Define body of foo here
}
```

Der Prototyp muss den Rückgabetyp (`void`), den Namen der Funktion (`foo`) und die `foo` der Argumentliste (`int`) angeben. Die Namen der Argumente sind jedoch NICHT erforderlich .

Eine gängige Möglichkeit, dies in die Organisation von Quelldateien zu integrieren, besteht darin, eine Header-Datei mit allen Prototypdeklarationen zu erstellen:

```
// foo.h
void foo(int); // prototype declaration
```

und dann die vollständige Definition an anderer Stelle angeben:

```
// foo.cpp --> foo.o
#include "foo.h" // foo's prototype declaration is "hidden" in here
void foo(int x) { } // foo's body definition
```

und verknüpfen `foo.o` nach dem Kompilieren die entsprechende Objektdatei `foo.o` mit der

kompilierten Objektdatei, in der sie in der Verknüpfungsphase verwendet wird, `main.o` :

```
// main.cpp --> main.o
#include "foo.h" // foo's prototype declaration is "hidden" in here
int main() { foo(2); } // foo is valid to call because its prototype declaration was
beforehand.
// the prototype and body definitions of foo are linked through the object files
```

Ein „nicht aufgelöstes externes Symbol“ -Fehler auftritt , wenn der *Funktionsprototyp* und *Anruf* existieren, aber der *Funktionskörper* ist nicht definiert. Das Auflösen kann schwieriger sein, da der Compiler den Fehler erst im letzten Verknüpfungsschritt meldet und nicht weiß, in welche Zeile im Code gesprungen werden soll, um den Fehler anzuzeigen.

Der Standard-C ++ - Kompilierungsprozess

Ausführbarer C ++ - Programmcode wird normalerweise von einem Compiler erzeugt.

Ein **Compiler** ist ein Programm, das Code aus einer Programmiersprache in eine andere Form übersetzt, die für einen Computer (mehr) direkt ausführbar ist. Die Verwendung eines Compilers zum Übersetzen von Code wird als **Kompilierung bezeichnet.**

C ++ erbt die Form des Kompilierungsprozesses von seiner "übergeordneten" Sprache C. Nachfolgend finden Sie eine Liste mit den vier Hauptschritten der Kompilierung in C ++:

1. Der C ++ - Präprozessor kopiert den Inhalt aller enthaltenen Header-Dateien in die Quellcodedatei, generiert Makrocode und ersetzt symbolische Konstanten, die mit #define definiert sind, durch ihre Werte.
2. Die vom C ++ - Präprozessor erzeugte erweiterte Quellcodedatei wird in eine für die Plattform geeignete Assemblersprache kompiliert.
3. Der vom Compiler generierte Assembler-Code wird zu einem geeigneten Objektcode für die Plattform zusammengestellt.
4. Die vom Assembler generierte Objektcodedatei wird mit den Objektcodedateien für alle Bibliotheksfunktionen verknüpft, die zum Erstellen einer ausführbaren Datei verwendet werden.

- Hinweis: Ein Teil des kompilierten Codes ist miteinander verknüpft, jedoch nicht zum Erstellen eines endgültigen Programms. Normalerweise kann dieser "verknüpfte" Code auch in ein Format gepackt werden, das von anderen Programmen verwendet werden kann. Dieses "Bündel von verpacktem, verwendbarem Code" bezeichnen C ++ - Programmierer als **Bibliothek.**

Viele C ++ - Compiler können bestimmte Teile des Kompilierungsprozesses zur Vereinfachung oder zur zusätzlichen Analyse zusammenführen oder deren Zusammenführung aufheben. Viele C ++ - Programmierer verwenden verschiedene Werkzeuge, aber alle Werkzeuge folgen im Allgemeinen diesem allgemeinen Prozess, wenn sie an der Erstellung eines Programms beteiligt sind.

Der Link unten erweitert diese Diskussion und bietet eine schöne Grafik, um zu helfen. [1]: http://faculty.cs.niu.edu/~mcmahon/CS241/Notes/compile.html

Präprozessor

Der Präprozessor ist ein wichtiger Teil des Compilers.

Es bearbeitet den Quellcode, schneidet einige Bits heraus, ändert andere und fügt andere Dinge hinzu.

In Quelldateien können Präprozessoranweisungen eingefügt werden. Diese Anweisungen weisen den Präprozessor an, bestimmte Aktionen auszuführen. Eine Direktive beginnt mit einem # in einer neuen Zeile. Beispiel:

```
#define ZERO 0
```

Die erste Präprozessoranweisung, die Sie treffen werden, ist wahrscheinlich die

```
#include <something>
```

Richtlinie. Was sie tut , ist nimmt alle `something` und fügt es in der Datei , wenn die Richtlinie war. Das Hallo Weltprogramm beginnt mit der Zeile

```
#include <iostream>
```

In dieser Zeile werden die Funktionen und Objekte hinzugefügt, mit denen Sie die Standardeingabe und -ausgabe verwenden können.

Die C-Sprache, die auch den Präprozessor verwendet, hat nicht so viele Header-Dateien wie die C ++ - Sprache, aber in C ++ können Sie alle C-Header-Dateien verwenden.

Die nächste wichtige Richtlinie ist wahrscheinlich die

```
#define something something_else
```

Richtlinie. Dies teilt dem Präprozessor mit, dass er beim Durchlaufen der Datei jedes Vorkommen von `something` mit `something_else` ersetzen sollte. Es kann auch Dinge ähnlich wie Funktionen machen, aber das zählt wahrscheinlich als fortgeschrittenes C ++.

Die `something_else` wird nicht benötigt, aber wenn Sie `something` als nichts definieren, dann verschwinden außerhalb der Präprozessoranweisungen alle Vorkommen von `something` .

Dies ist aufgrund der Direktiven `#if` , `#else` und `#ifdef` tatsächlich hilfreich. Das Format für diese wäre folgendes:

```
#if something==true
//code
#else
//more code
#endif
```

```
#ifdef thing_that_you_want_to_know_if_is_defined
//code
#endif
```

Diese Anweisungen fügen den Code ein, der sich im True-Bit befindet, und löschen die False-Bits. Dies kann verwendet werden, um Codebits zu haben, die nur in bestimmten Betriebssystemen enthalten sind, ohne dass der gesamte Code neu geschrieben werden muss.

Erste Schritte mit C ++ online lesen: https://riptutorial.com/de/cplusplus/topic/206/erste-schritte-mit-c-plusplus

Kapitel 2: Ablaufsteuerung

Bemerkungen

Schauen Sie sich das Thema Schleifen für die verschiedenen Arten von Schleifen an.

Examples

Fall

Führt eine Fallbezeichnung einer switch-Anweisung ein. Der Operand muss ein konstanter Ausdruck sein und mit der Schalterbedingung in type übereinstimmen. Wenn die switch-Anweisung ausgeführt wird, springt sie zu der Fallbezeichnung, wobei der Operand der Bedingung entspricht, falls vorhanden.

```
char c = getchar();
bool confirmed;
switch (c) {
  case 'y':
    confirmed = true;
    break;
  case 'n':
    confirmed = false;
    break;
  default:
    std::cout << "invalid response!\n";
    abort();
}
```

Schalter

Gemäß dem C ++ - Standard

> Die `switch` Anweisung bewirkt, dass die Steuerung in Abhängigkeit von dem Wert einer Bedingung an eine von mehreren Anweisungen übergeben wird.

Der Schlüsselwort - `switch` wird von einem geklammerten Zustand und einem Block folgt, der enthalten kann `case` Etikett und ein optionales `default` Wenn der Schalter - Anweisung ausgeführt wird, wird die Steuerung entweder auf einen übertragen `case` Etikett mit einem Wert, der dem Zustand, falls vorhanden, oder an die passenden `default` - Etikett, falls vorhanden.

Die Bedingung muss ein Ausdruck oder eine Deklaration sein, die entweder einen Ganzzahl- oder Aufzählungstyp hat, oder ein Klassentyp mit einer Konvertierungsfunktion in Ganzzahl oder Aufzählungstyp.

```
char c = getchar();
bool confirmed;
switch (c) {
```

```
  case 'y':
    confirmed = true;
    break;
  case 'n':
    confirmed = false;
    break;
  default:
    std::cout << "invalid response!\n";
    abort();
}
```

Fang

Das `catch` Schlüsselwort führt einen Ausnahmehandler ein, d. H. Einen Block, in den das Steuerelement übertragen wird, wenn eine Ausnahme des kompatiblen Typs ausgelöst wird. Auf das `catch` Schlüsselwort folgt eine in Klammern stehende *Ausnahmedeklaration* , die in ihrer Form einer Funktionsparameterdeklaration ähnelt: Der Parametername kann weggelassen werden, und die Auslassungszeichen `...` sind zulässig, was jedem Typ entspricht. Der Ausnahmehandler behandelt die Ausnahmebedingung nur, wenn ihre Deklaration mit dem Ausnahmetyp kompatibel ist. Weitere Informationen finden Sie unter Ausnahmen abfangen .

```
try {
    std::vector<int> v(N);
    // do something
} catch (const std::bad_alloc&) {
    std::cout << "failed to allocate memory for vector!" << std::endl;
} catch (const std::runtime_error& e) {
    std::cout << "runtime error: " << e.what() << std::endl;
} catch (...) {
    std::cout << "unexpected exception!" << std::endl;
    throw;
}
```

Standard

Führt in einer switch-Anweisung eine Beschriftung ein, zu der gesprungen wird, wenn der Wert der Bedingung nicht mit den Werten der Fallbezeichnungen übereinstimmt.

```
char c = getchar();
bool confirmed;
switch (c) {
  case 'y':
    confirmed = true;
    break;
  case 'n':
    confirmed = false;
    break;
  default:
    std::cout << "invalid response!\n";
    abort();
}
```

C ++ 11

Definiert einen Standardkonstruktor, einen Kopierkonstruktor, einen Verschiebungskonstruktor, einen Destruktor, einen Kopierzuweisungsoperator oder einen Verschiebungszuweisungsoperator, um sein Standardverhalten zu erhalten.

```
class Base {
    // ...
    // we want to be able to delete derived classes through Base*,
    // but have the usual behaviour for Base's destructor.
    virtual ~Base() = default;
};
```

ob

Führt eine if-Anweisung ein. Dem Schlüsselwort `if` muss eine in Klammern stehende Bedingung folgen, die entweder ein Ausdruck oder eine Deklaration sein kann. Wenn die Bedingung wahr ist, wird die Unteranweisung nach der Bedingung ausgeführt.

```
int x;
std::cout << "Please enter a positive number." << std::endl;
std::cin >> x;
if (x <= 0) {
    std::cout << "You didn't enter a positive number!" << std::endl;
    abort();
}
```

sonst

Auf die erste Unteranweisung einer if-Anweisung kann das Schlüsselwort `else` folgen. Die Unteranweisung nach dem Schlüsselwort `else` wird ausgeführt, wenn die Bedingung falsch ist (dh wenn die erste Unteranweisung nicht ausgeführt wird).

```
int x;
std::cin >> x;
if (x%2 == 0) {
    std::cout << "The number is even\n";
} else {
    std::cout << "The number is odd\n";
}
```

gehe zu

Springt zu einer beschrifteten Anweisung, die sich in der aktuellen Funktion befinden muss.

```
bool f(int arg) {
    bool result = false;
    hWidget widget = get_widget(arg);
    if (!g()) {
        // we can't continue, but must do cleanup still
        goto end;
    }
    // ...
    result = true;
```

```
  end:
    release_widget(widget);
    return result;
}
```

Rückkehr

Gibt die Kontrolle von einer Funktion an ihren Aufrufer zurück.

Wenn `return` einen Operanden hat, wird der Operand in den Rückgabetyp der Funktion konvertiert und der konvertierte Wert wird an den Aufrufer zurückgegeben.

```
int f() {
    return 42;
}
int x = f(); // x is 42
int g() {
    return 3.14;
}
int y = g(); // y is 3
```

Wenn `return` keinen Operanden hat, muss die Funktion einen `void` Rückgabetyp haben. Als Sonderfall kann eine `void` -returning-Funktion auch einen Ausdruck zurückgeben, wenn der Typ den Typ `void` .

```
void f(int x) {
    if (x < 0) return;
    std::cout << sqrt(x);
}
int g() { return 42; }
void h() {
    return f(); // calls f, then returns
    return g(); // ill-formed
}
```

Wenn `main` zurückkehrt, wird `std::exit` implizit mit dem Rückgabewert aufgerufen, und der Wert wird somit an die Ausführungsumgebung zurückgegeben. (Das Zurückkehren von `main` zerstört jedoch automatisch lokale Variablen, während `std::exit` direkt aufgerufen wird.)

```
int main(int argc, char** argv) {
    if (argc < 2) {
        std::cout << "Missing argument\n";
        return EXIT_FAILURE; // equivalent to: exit(EXIT_FAILURE);
    }
}
```

werfen

1. Wenn `throw` in einem Ausdruck mit einem Operanden auftritt, ist seine Wirkung eine werfen Ausnahme , die eine Kopie des Operanden ist.

   ```
   void print_asterisks(int count) {
   ```

```
    if (count < 0) {
        throw std::invalid_argument("count cannot be negative!");
    }
    while (count--) { putchar('*'); }
}
```

2. Wenn `throw` ohne einen Operanden in einem Ausdruck auftritt, ist seine Wirkung auf die aktuelle Ausnahme erneut auslösen . Wenn keine aktuelle Ausnahme `std::terminate` , wird `std::terminate` aufgerufen.

```
try {
    // something risky
} catch (const std::bad_alloc&) {
    std::cerr << "out of memory" << std::endl;
} catch (...) {
    std::cerr << "unexpected exception" << std::endl;
    // hope the caller knows how to handle this exception
    throw;
}
```

3. Wenn der `throw` in einem Funktionsdeklarator auftritt, führt er eine dynamische Ausnahmespezifikation ein, die die Typen von Ausnahmen auflistet, die die Funktion weitergeben darf.

```
// this function might propagate a std::runtime_error,
// but not, say, a std::logic_error
void risky() throw(std::runtime_error);
// this function can't propagate any exceptions
void safe() throw();
```

   Dynamische Ausnahmespezifikationen werden ab C ++ 11 nicht mehr unterstützt.

Beachten Sie, dass die ersten beiden oben genannten Verwendungsarten von `throw` Ausdrücke und keine Anweisungen darstellen. (Der Typ eines Wurfausdrucks ist `void` .) Dadurch können sie innerhalb von Ausdrücken verschachtelt werden, wie dies z.

```
unsigned int predecessor(unsigned int x) {
    return (x > 0) ? (x - 1) : (throw std::invalid_argument("0 has no predecessor"));
}
```

Versuchen

Auf das Schlüsselwort `try` folgt ein Block oder eine Konstruktor-Initialisierungsliste und dann ein Block (siehe hier). Auf den try-Block folgen ein oder mehrere Fangblöcke . Wenn sich eine Ausnahme aus dem try-Block verbreitet, hat jeder der entsprechenden catch-Blöcke nach dem try-Block die Möglichkeit, die Ausnahme zu behandeln, wenn die Typen übereinstimmen.

```
std::vector<int> v(N);     // if an exception is thrown here,
                           // it will not be caught by the following catch block
try {
    std::vector<int> v(N); // if an exception is thrown here,
```

```
                            // it will be caught by the following catch block
    // do something with v
} catch (const std::bad_alloc&) {
    // handle bad_alloc exceptions from the try block
}
```

Bedingte Strukturen: if, if..else

wenn und sonst:

es wurde verwendet, um zu überprüfen, ob der angegebene Ausdruck wahr oder falsch zurückgibt und als solcher wirkt:

```
if (condition) statement
```

Die Bedingung kann ein beliebiger gültiger C ++ - Ausdruck sein, der etwas zurückgibt, das beispielsweise mit der Wahrheit / Falschheit geprüft wird:

```
if (true) { /* code here */ }  // evaluate that true is true and execute the code in the
brackets
if (false) { /* code here */ } // always skip the code since false is always false
```

Die Bedingung kann beispielsweise alles sein, eine Funktion, eine Variable oder ein Vergleich

```
if(istrue()) { } // evaluate the function, if it returns true, the if will execute the code
if(isTrue(var)) { } //evalute the return of the function after passing the argument var
if(a == b) { } // this will evaluate the return of the experssion (a==b) which will be true if
equal and false if unequal
if(a) { } //if a is a boolean type, it will evaluate for its value, if it's an integer, any
non zero value will be true,
```

Wenn Sie nach mehreren Ausdrücken suchen möchten, haben Sie zwei Möglichkeiten:

mit binären Operatoren :

```
if (a && b) { } // will be true only if both a and b are true (binary operators are outside
the scope here
if (a || b ) { } //true if a or b is true
```

mit if / ifelse / else :

für einen einfachen Schalter entweder wenn oder sonst

```
if (a== "test") {
    //will execute if a is a string "test"
} else {
    // only if the first failed, will execute
}
```

für mehrere Auswahlmöglichkeiten:

```
if (a=='a') {
// if a is a char valued 'a'
} else if (a=='b') {
// if a is a char valued 'b'
} else if (a=='c') {
// if a is a char valued 'c'
} else {
//if a is none of the above
}
```

Es muss jedoch beachtet werden, dass Sie stattdessen ' **switch** ' verwenden sollten, wenn Ihr Code nach dem Wert dieser Variablen sucht

Jump-Anweisungen: Pause, Weiter, Los, Beenden.

Die Pause-Anweisung:

Mit break können wir eine Schleife verlassen, auch wenn die Bedingung für das Ende nicht erfüllt ist. Es kann verwendet werden, um eine Endlosschleife zu beenden oder sie vor ihrem natürlichen Ende zu beenden

Die Syntax lautet

```
break;
```

Beispiel : Wir verwenden häufig `break` in `switch` Fällen, dh wenn ein switch- `switch` einmal erfüllt ist, wird der Codeblock dieser Bedingung ausgeführt.

```
switch(conditon){
case 1: block1;
case 2: block2;
case 3: block3;
default: blockdefault;
}
```

Wenn der Fall 1 erfüllt ist, wird in diesem Fall Block 1 ausgeführt. Was wir wirklich wollen, ist, dass nur der Block 1 verarbeitet wird. Wenn der Block 1 jedoch einmal verarbeitet wird, werden die Blöcke Block2, Block3 und Blockdefault ebenfalls verarbeitet, obwohl nur der Fall 1 erfüllt wurde Um dies zu vermeiden, verwenden wir break am Ende jedes Blocks wie:

```
switch(condition){
case 1: block1;
        break;
case 2: block2;
        break;
case 3: block3;
        break;
default: blockdefault;
        break;
}
```

Es wird also nur ein Satz bearbeitet und die Steuerung verlässt die Schalterschleife.

break kann auch in anderen bedingten und nicht bedingten Schleifen verwendet werden, z. B. `if` , `while` , `for` etc;

Beispiel:

```
if(condition1){
   ....
   if(condition2){
    .......
    break;
    }
 ...
}
```

Die weitere Anweisung:

Die continue-Anweisung bewirkt, dass das Programm den Rest der Schleife in der aktuellen Iteration überspringt, als wäre das Ende des Anweisungsblocks erreicht, wodurch es zur folgenden Iteration springt.

Die Syntax lautet

```
continue;
```

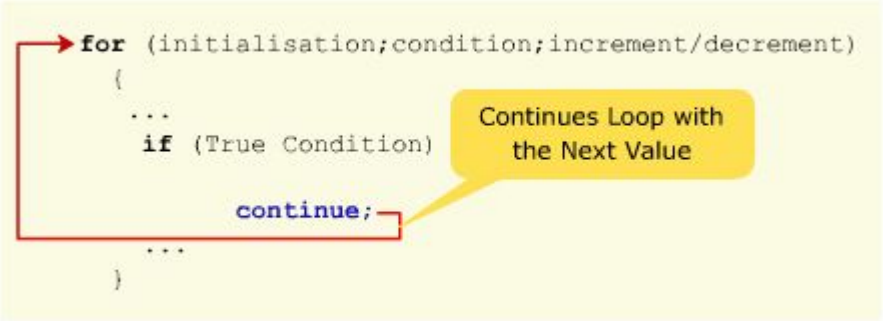
Beispiel betrachten Sie Folgendes:

```
for(int i=0;i<10;i++){
if(i%2==0)
continue;
cout<<"\n @"<<i;
}
```

was die Ausgabe erzeugt:

```
 @1
 @3
 @5
 @7
 @9
```

Wenn dieser Code immer dann erfüllt ist, wenn die Bedingung erfüllt ist, dass der Befehl " `i%2==0 continue` wird, wird der gesamte verbleibende Code übersprungen (@ und i), und die Inkrement- / Dekrementierungsanweisung der Schleife wird ausgeführt.



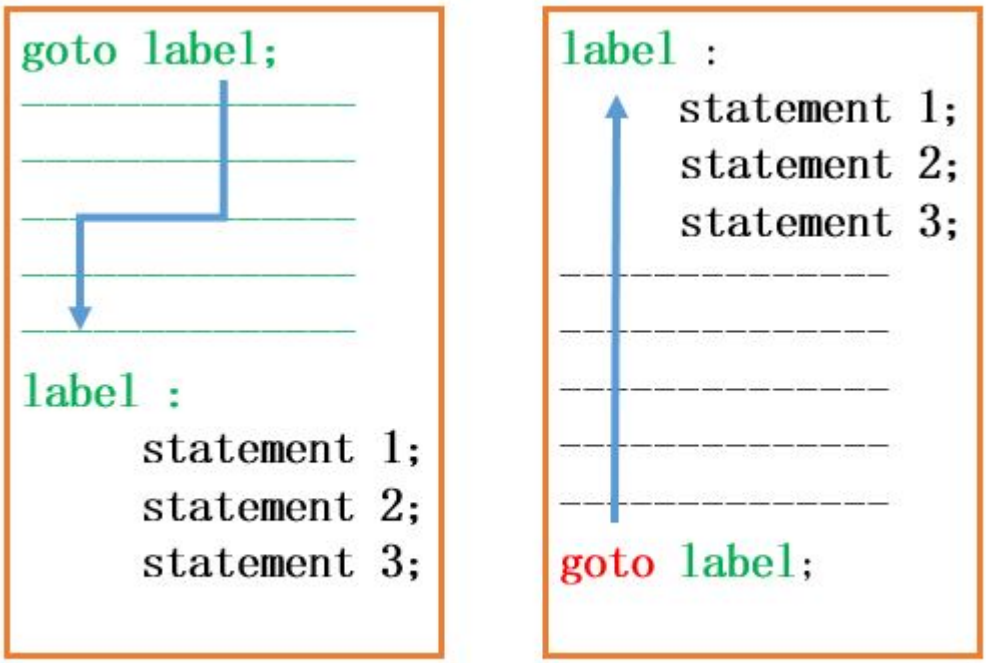
Die goto Anweisung:

Es ermöglicht einen absoluten Sprung zu einem anderen Punkt im Programm. Sie sollten diese Funktion sorgfältig verwenden, da bei der Ausführung keine Einschränkung der Verschachtelung auftritt. Der Zielpunkt wird durch ein Label identifiziert, das dann als Argument für die goto-Anweisung verwendet wird. Ein Label besteht aus einem gültigen Bezeichner gefolgt von einem Doppelpunkt (:)

Die Syntax lautet

```
goto label;
..
.
label: statement;
```

Hinweis: Von der *Verwendung der goto-Anweisung wird dringend abgeraten, da es schwierig ist, den Steuerfluss eines Programms zu verfolgen, wodurch das Programm schwer verständlich und schwer zu ändern ist.*



Forward Reference Backward Reference

Beispiel:

```
int num = 1;
STEP:
do{

    if( num%2==0 )
    {
        num = num + 1;
        goto STEP;
     }

   cout << "value of num : " << num << endl;
   num = num + 1;
 }while( num < 10 );
```

Ausgabe :

```
value of num : 1
value of num : 3
value of num : 5
value of num : 7
value of num : 9
```

Immer wenn die Bedingung `num%2==0` erfüllt ist, sendet der goto die Ausführungssteuerung an den Anfang der `do-while` Schleife.

Die Exit-Funktion:

`exit` ist eine in `cstdlib` definierte `cstdlib` . Der Zweck des `exit` besteht darin, das laufende Programm mit einem bestimmten Exitcode zu beenden. Sein Prototyp ist:

```
void exit (int exit code);
```

`cstdlib` definiert die Standard-Exit-Codes `EXIT_SUCCESS` und `EXIT_FAILURE` .

Ablaufsteuerung online lesen: https://riptutorial.com/de/cplusplus/topic/7837/ablaufsteuerung

Kapitel 3: Argumentabhängige Namenssuche

Examples

Welche Funktionen gefunden werden

Funktionen werden gefunden, indem zuerst eine Gruppe von *"verknüpften Klassen"* und *"verknüpften Namespaces" erfasst wird* , die je nach Argumenttyp `T` einen oder mehrere der folgenden `T` . Lassen Sie uns zunächst die Regeln für die Klassennamen von Klassen, Enumeration und Klassenvorlagen anzeigen.

- Wenn `T` eine verschachtelte Klasse ist, Member-Enumeration, dann die umgebende Klasse.
- Wenn `T` eine Aufzählung ist (es kann *auch* ein Klassenmitglied sein!), Der innerste Namespace davon.
- Wenn `T` eine Klasse ist (kann *auch* geschachtelt sein!), Alle ihre Basisklassen und die Klasse selbst. Der innerste Namespace aller zugehörigen Klassen.
- Wenn `T` ein `ClassTemplate<TemplateArguments>` (dies ist *auch* eine Klasse!), Die mit den Vorlagentypargumenten verknüpften Klassen und Namespaces, der Namensraum eines Vorlagenvorlagenarguments und die umgebende Klasse eines Vorlagenvorlagenarguments, sofern ein Vorlagenargument vorhanden ist eine Mitgliedsvorlage

Nun gibt es auch ein paar Regeln für eingebaute Typen

- Wenn `T` ein Zeiger auf `U` oder ein Array von `U` , die mit `U` verknüpften Klassen und Namespaces. Beispiel: `void (*fptr)(A); f(fptr);` , enthält die mit `void(A)` verknüpften Namespaces und Klassen (siehe nächste Regel).
- Wenn `T` ein Funktionstyp ist, die mit Parameter- und Rückgabetypen verknüpften Klassen und Namespaces. Beispiel: `void(A)` enthält die mit `A` verknüpften Namespaces und Klassen.
- Wenn `T` ein Zeiger auf einen Member ist, werden die Klassen und Namespaces, die mit dem Member-Typ verknüpft sind (möglicherweise auf Zeiger auf Member-Funktionen und Zeiger auf Datenmitglied!) Angewendet. Beispiel: `BA::*p; void (A::*pf)(B); f(p); f(pf);` schließt die mit `A` , `B` , `void(B)` verknüpften Namespaces und Klassen `A` (wobei das Aufzählungszeichen oben für Funktionstypen gilt).

Alle Funktionen und Vorlagen in allen zugehörigen Namespaces werden durch eine argumentabhängige Suche gefunden. Außerdem werden *in zugehörigen Klassen deklarierte Freundesfunktionen für den Namensbereich gefunden* , die normalerweise nicht sichtbar sind. Die Verwendung von Direktiven wird jedoch ignoriert.

Alle folgenden Beispielaufrufe sind gültig, ohne `f` anhand des Namespace-Namens im Aufruf zu qualifizieren.

```
namespace A {
   struct Z { };
   namespace I { void g(Z); }
   using namespace I;
```

```
   struct X { struct Y { }; friend void f(Y) { } };
   void f(X p) { }
   void f(std::shared_ptr<X> p) { }
}

// example calls
f(A::X());
f(A::X::Y());
f(std::make_shared<A::X>());

g(A::Z()); // invalid: "using namespace I;" is ignored!
```

Argumentabhängige Namenssuche online lesen:
https://riptutorial.com/de/cplusplus/topic/5163/argumentabhangige-namenssuche

Kapitel 4: Arithmitische Metaprogrammierung

Einführung

Dies ist ein Beispiel für die Verwendung der C ++ - Template-Metaprogrammierung bei der Verarbeitung von Rechenoperationen in Kompilierzeit.

Examples

Rechenleistung in O (log n)

Dieses Beispiel zeigt eine effiziente Methode zur Berechnung der Leistung mithilfe der Metaprogrammierung von Vorlagen.

```
template <int base, unsigned int exponent>
struct power
{
    static const int halfvalue = power<base, exponent / 2>::value;
    static const int value = halfvalue * halfvalue * power<base, exponent % 2>::value;
};

template <int base>
struct power<base, 0>
{
   static const int value = 1;
   static_assert(base != 0, "power<0, 0> is not allowed");
};


template <int base>
struct power<base, 1>
{
    static const int value = base;
};
```

Verwendungsbeispiel:

```
std::cout << power<2, 9>::value;
```

C ++ 14

Dieser behandelt auch negative Exponenten:

```
template <int base, int exponent>
struct powerDouble
{
    static const int exponentAbs = exponent < 0 ? (-exponent) : exponent;
    static const int halfvalue = powerDouble<base, exponentAbs / 2>::intermediateValue;
    static const int intermediateValue = halfvalue * halfvalue * powerDouble<base, exponentAbs
% 2>::intermediateValue;
```

```
    constexpr static double value = exponent < 0 ? (1.0 / intermediateValue) :
intermediateValue;

};

template <int base>
struct powerDouble<base, 0>
{
    static const int intermediateValue = 1;
    constexpr static double value = 1;
    static_assert(base != 0, "powerDouble<0, 0> is not allowed");
};


template <int base>
struct powerDouble<base, 1>
{
    static const int intermediateValue = base;
    constexpr static double value = base;
};


int main()
{
    std::cout << powerDouble<2,-3>::value;
}
```

Arithmitische Metaprogrammierung online lesen:
https://riptutorial.com/de/cplusplus/topic/10907/arithmitische-metaprogrammierung

Kapitel 5: Arrays

Einführung

Arrays sind Elemente desselben Typs, die sich in benachbarten Speicherstellen befinden. Die Elemente können einzeln durch einen eindeutigen Bezeichner mit einem hinzugefügten Index referenziert werden.

Auf diese Weise können Sie mehrere Variablenwerte eines bestimmten Typs deklarieren und einzeln darauf zugreifen, ohne für jeden Wert eine Variable deklarieren zu müssen.

Examples

Arraygröße: Typ sicher zur Kompilierzeit.

```
#include <stddef.h>     // size_t, ptrdiff_t

//---------------------------------- Machinery:

using Size = ptrdiff_t;

template< class Item, size_t n >
constexpr auto n_items( Item (&)[n] ) noexcept
    -> Size
{ return n; }


//---------------------------------- Usage:

#include <iostream>
using namespace std;
auto main()
    -> int
{
    int const   a[]     = {3, 1, 4, 1, 5, 9, 2, 6, 5, 4};
    Size const  n       = n_items( a );
    int         b[n]    = {};       // An array of the same size as a.

    (void) b;
    cout << "Size = " << n << "\n";
}
```

Das C-Idiom für die Array-Größe `sizeof(a)/sizeof(a[0])` akzeptiert einen Zeiger als Argument und liefert im Allgemeinen ein falsches Ergebnis.

Für C ++ 11

Mit C ++ 11 können Sie Folgendes tun:

```
std::extent<decltype(MyArray)>::value;
```

Beispiel:

```
char MyArray[] = { 'X','o','c','e' };
const auto n = std::extent<decltype(MyArray)>::value;
std::cout << n << "\n"; // Prints 4
```

Bis 17 ++ C (bevorstehend zum Zeitpunkt des Schreibens) C ++ hatte keine Einbau-Kernsprache oder Standardbibliothek Dienstprogramm die Größe eines Arrays zu erhalten, aber dies kann durch Hindurchleiten der Array *unter Bezugnahme* auf eine Funktionsschablone implementiert werden, wie oben gezeigt. Fein, aber wichtig: Der Parameter für die Vorlagengröße ist ein `size_t` , der mit dem Ergebnistyp der signierten `Size` Funktion etwas inkonsistent ist, um den g ++ - Compiler unterzubringen, der manchmal auf `size_t` für den Vorlagenabgleich besteht.

Mit C ++ 17 und höher kann man stattdessen `std::size` , das auf Arrays spezialisiert ist.

Rohes Array mit dynamischer Größe

```
// Example of raw dynamic size array. It's generally better to use std::vector.
#include <algorithm>            // std::sort
#include <iostream>
using namespace std;

auto int_from( istream& in ) -> int { int x; in >> x; return x; }

auto main()
    -> int
{
    cout << "Sorting n integers provided by you.\n";
    cout << "n? ";
    int const   n   = int_from( cin );
    int*        a   = new int[n];       // ← Allocation of array of n items.

    for( int i = 1; i <= n; ++i )
    {
        cout << "The #" << i << " number, please: ";
        a[i-1] = int_from( cin );
    }

    sort( a, a + n );
    for( int i = 0; i < n; ++i ) { cout << a[i] << ' '; }
    cout << '\n';

    delete[] a;
}
```

Ein Programm, das ein Array `T a[n];` deklariert `T a[n];` *Wenn* `n` zur Laufzeit bestimmt wird, kann es mit bestimmten Compilern kompiliert werden, die C99- *Arrays mit variadischer Länge* (VLAs) als Spracherweiterung unterstützen. VLAs werden jedoch nicht von Standard C ++ unterstützt. Dieses Beispiel zeigt, wie Sie ein Array mit dynamischer Größe manuell über einen `new[]` - Expression zuweisen können.

```
int*        a   = new int[n];       // ← Allocation of array of n items.
```

… Dann verwenden Sie es und heben Sie es schließlich über einen `delete[]` -Expression auf:

```
delete[] a;
```

Das hier zugewiesene Array hat unbestimmte Werte, es kann jedoch durch einfaches Hinzufügen einer leeren Klammer `()` null initialisiert werden: `new int[n]()` . Im Allgemeinen führt dies für einen beliebigen Elementtyp eine *Wertinitialisierung durch* .

Als Teil einer Funktion in einer Aufrufhierarchie wäre dieser Code nicht ausnahmesicher, da eine Ausnahme vor dem Ausdruck `delete[]` (und nach dem `new[]`) einen Speicherverlust verursachen würde. Eine Möglichkeit, dieses Problem anzugehen, ist die Automatisierung der Bereinigung, z. B. über einen intelligenten Zeiger `std::unique_ptr` . Im Allgemeinen ist es jedoch eine bessere `std::vector` , einfach einen `std::vector` zu verwenden. Dafür gibt es `std::vector` .

Erweitern des Arrays dynamischer Größe mithilfe von std :: vector.

```
// Example of std::vector as an expanding dynamic size array.
#include <algorithm>            // std::sort
#include <iostream>
#include <vector>               // std::vector
using namespace std;

int int_from( std::istream& in ) { int x = 0; in >> x; return x; }

int main()
{
    cout << "Sorting integers provided by you.\n";
    cout << "You can indicate EOF via F6 in Windows or Ctrl+D in Unix-land.\n";
    vector<int> a;      // ← Zero size by default.

    while( cin )
    {
        cout << "One number, please, or indicate EOF: ";
        int const x = int_from( cin );
        if( !cin.fail() ) { a.push_back( x ); }  // Expands as necessary.
    }

    sort( a.begin(), a.end() );
    int const n = a.size();
    for( int i = 0; i < n; ++i ) { cout << a[i] << ' '; }
    cout << '\n';
}
```

`std::vector` ist eine Standardvorlage für Bibliotheksklassen, die die Vorstellung eines Arrays mit variabler Größe bereitstellt. Es kümmert sich um die gesamte Speicherverwaltung, und der Puffer ist zusammenhängend, so dass ein Zeiger auf den Puffer (z. B. `&v[0]` oder `v.data()`) an API-Funktionen übergeben werden kann, die ein Raw-Array erfordern. Ein `vector` kann sogar zur Laufzeit erweitert werden, z. B. über die `push_back` push_back, die ein Element anfügt.

Die Komplexität der Folge von *n* `push_back` Operationen, einschließlich des Kopierens oder Verschiebens der `push_back` , wird amortisiert O (*n*). Amortisiert: im Durchschnitt.

Intern wird dies normalerweise dadurch erreicht, dass der Vektor seine Puffergröße und Kapazität

verdoppelt , wenn ein größerer Puffer benötigt wird. Wenn ein Puffer beispielsweise mit der Größe 1 beginnt und bei *n* = 17 `push_back` Aufrufen wiederholt verdoppelt wird, sind 1 + 2 + 4 + 8 + 16 = 31 Kopiervorgänge erforderlich, was weniger als 2 × *n* = 34 ist Im Allgemeinen kann die Summe dieser Sequenz 2 × *n* nicht überschreiten.

Verglichen mit dem Beispiel für ein dynamisches Array mit dynamischer Größe erfordert dieser `vector` Code nicht, dass der Benutzer die Anzahl der Elemente vorab bereitstellt (und kennt) muss. Stattdessen wird der Vektor für jeden neuen vom Benutzer angegebenen Elementwert nur nach Bedarf erweitert.

Eine Raw-Array-Matrix mit fester Größe (d. H. Ein 2D-Raw-Array).

```
// A fixed size raw array matrix (that is, a 2D raw array).
#include <iostream>
#include <iomanip>
using namespace std;

auto main() -> int
{
    int const   n_rows  = 3;
    int const   n_cols  = 7;
    int const   m[n_rows][n_cols] =             // A raw array matrix.
    {
        {  1,  2,  3,  4,  5,  6,  7 },
        {  8,  9, 10, 11, 12, 13, 14 },
        { 15, 16, 17, 18, 19, 20, 21 }
    };

    for( int y = 0; y < n_rows; ++y )
    {
        for( int x = 0; x < n_cols; ++x )
        {
            cout << setw( 4 ) << m[y][x];       // Note: do NOT use m[y,x]!
        }
        cout << '\n';
    }
}
```

Ausgabe:

```
   1   2   3   4   5   6   7
   8   9  10  11  12  13  14
  15  16  17  18  19  20  21
```

C ++ unterstützt keine spezielle Syntax für die Indizierung eines mehrdimensionalen Arrays. Stattdessen wird ein solches Array als Array von Arrays (möglicherweise von Arrays usw.) betrachtet, und für jede Ebene wird die normale Einzelindexnotation `[ i ]` verwendet. Im obigen Beispiel bezieht sich `m[y]` auf die Zeile `y` von `m` , wobei `y` ein auf Null basierender Index ist. Dann kann diese Zeile wiederum indiziert werden, z `m[y][x]` B. `m[y][x]` , was sich auf das `x` te Element oder die Spalte von Zeile `y` bezieht.

Dh der letzte Index variiert am schnellsten, und in der Deklaration ist der Bereich dieses Indexes, der hier die Anzahl der Spalten pro Zeile ist, die letzte und "innerste" Größe.

Da C ++ keine dynamische Unterstützung für Arrays mit dynamischer Größe bietet, wird die dynamische Matrix, außer der dynamischen Zuweisung, häufig als Klasse implementiert. Dann hat die Indizierungsnotation für die rohe Array-Matrix `m[y][x]` einige Kosten, entweder durch Offenlegung der Implementierung (sodass z. B. die Ansicht einer transponierten Matrix praktisch unmöglich wird) oder durch Hinzufügen eines zusätzlichen Aufwands und geringfügiger Unbequemlichkeiten, wenn sie ausgeführt werden ein Proxy-Objekt von `operator[]` . Die Indizierungsnotation für eine solche Abstraktion kann und wird daher normalerweise unterschiedlich sein, sowohl im Look & Feel als auch in der Reihenfolge der Indizes, z `m(x,y)` B. `m(x,y)` oder `m.at(x,y)` oder `m.item(x,y)` .

Eine dynamische Größenmatrix unter Verwendung von std :: vector zur Speicherung.

Leider gibt es seit C ++ 14 keine dynamische Größenmatrixklasse in der C ++ - Standardbibliothek. Matrix - Klassen , die dynamische Größe unterstützt , sind jedoch von einer Reihe von 3 rd Party - Bibliotheken, darunter der Boost - Matrix - Bibliothek (eine Unterbibliothek innerhalb der Boost - Bibliothek).

Wenn Sie keine Abhängigkeit von Boost oder einer anderen Bibliothek wünschen, ist die dynamische Größenmatrix eines armen Mannes in C ++ genau so

```
vector<vector<int>> m( 3, vector<int>( 7 ) );
```

... wo `vector` ist `std::vector` . Die Matrix wird hier erstellt, indem ein Zeilenvektor *n*- mal kopiert wird, wobei *n* die Anzahl der Zeilen, hier 3, ist. Sie hat den Vorteil, dass sie die gleiche `m[y][x]` - Indexnotation bereitstellt wie eine feste Array-Matrix mit fester Größe, jedoch Dies ist etwas ineffizient, da für jede Zeile eine dynamische Zuweisung erforderlich ist, und es ist ein bisschen unsicher, da eine unbeabsichtigte Größenänderung der Zeile möglich ist.

Ein sicherer und effizienter Ansatz besteht darin, einen einzelnen Vektor als *Speicher* für die Matrix zu verwenden und den Clientcode (*x* , *y*) einem entsprechenden Index in diesem Vektor zuzuordnen:

```
// A dynamic size matrix using std::vector for storage.

//---------------------------------------- Machinery:
#include <algorithm>        // std::copy
#include <assert.h>         // assert
#include <initializer_list> // std::initializer_list
#include <vector>           // std::vector
#include <stddef.h>         // ptrdiff_t

namespace my {
    using Size = ptrdiff_t;
    using std::initializer_list;
    using std::vector;

    template< class Item >
    class Matrix
    {
    private:
```

```
        vector<Item>    items_;
        Size            n_cols_;

        auto index_for( Size const x, Size const y ) const
            -> Size
        { return y*n_cols_ + x; }

    public:
        auto n_rows() const -> Size { return items_.size()/n_cols_; }
        auto n_cols() const -> Size { return n_cols_; }

        auto item( Size const x, Size const y )
            -> Item&
        { return items_[index_for(x, y)]; }

        auto item( Size const x, Size const y ) const
            -> Item const&
        { return items_[index_for(x, y)]; }

        Matrix(): n_cols_( 0 ) {}

        Matrix( Size const n_cols, Size const n_rows )
            : items_( n_cols*n_rows )
            , n_cols_( n_cols )
        {}

        Matrix( initializer_list< initializer_list<Item> > const& values )
            : items_()
            , n_cols_( values.size() == 0? 0 : values.begin()->size() )
        {
            for( auto const& row : values )
            {
                assert( Size( row.size() ) == n_cols_ );
                items_.insert( items_.end(), row.begin(), row.end() );
            }
        }
    };
}  // namespace my

//---------------------------------------- Usage:
using my::Matrix;

auto some_matrix()
    -> Matrix<int>
{
    return
    {
        {  1,  2,  3,  4,  5,  6,  7 },
        {  8,  9, 10, 11, 12, 13, 14 },
        { 15, 16, 17, 18, 19, 20, 21 }
    };
}

#include <iostream>
#include <iomanip>
using namespace std;
auto main() -> int
{
    Matrix<int> const m = some_matrix();
    assert( m.n_cols() == 7 );
    assert( m.n_rows() == 3 );
```

```
    for( int y = 0, y_end = m.n_rows(); y < y_end; ++y )
    {
        for( int x = 0, x_end = m.n_cols(); x < x_end; ++x )
        {
            cout << setw( 4 ) << m.item( x, y );        // ← Note: not `m[y][x]`!
        }
        cout << '\n';
    }
}
```

Ausgabe:

```
   1   2   3   4   5   6   7
   8   9  10  11  12  13  14
  15  16  17  18  19  20  21
```

Der obige Code ist nicht für die Industrie geeignet: Er soll die grundlegenden Prinzipien aufzeigen und den Bedürfnissen der Schüler entsprechen, die C ++ lernen.

Beispielsweise kann man `operator()` Überladungen definieren, um die Indizierungsnotation zu vereinfachen.

Array-Initialisierung

Ein Array ist nur ein Block von sequentiellen Speicherplätzen für einen bestimmten Variablentyp. Arrays werden auf dieselbe Weise wie normale Variablen zugewiesen, jedoch werden an ihren Namen `[]` eckige Klammern angehängt, die die Anzahl der Elemente enthalten, die in den Arrayspeicher passen.

Das folgende Beispiel eines Arrays verwendet den Typ `int` , den Variablennamen `arrayOfInts` und die Anzahl der Elemente `[5]` , für die das Array Platz hat:

```
int arrayOfInts[5];
```

Ein Array kann gleichzeitig deklariert und initialisiert werden

```
int arrayOfInts[5] = {10, 20, 30, 40, 50};
```

Bei der Initialisierung eines Arrays durch Auflisten aller seiner Mitglieder ist es nicht erforderlich, die Anzahl der Elemente in die eckigen Klammern einzufügen. Es wird automatisch vom Compiler berechnet. Im folgenden Beispiel sind es 5:

```
int arrayOfInts[] = {10, 20, 30, 40, 50};
```

Es ist auch möglich, nur die ersten Elemente zu initialisieren, während mehr Speicherplatz zugewiesen wird. In diesem Fall muss die Länge in Klammern definiert werden. Im Folgenden wird ein Array der Länge 5 mit Teilinitialisierung zugewiesen. Der Compiler initialisiert alle übrigen Elemente mit dem Standardwert des Elementtyps, in diesem Fall Null.

```
int arrayOfInts[5] = {10,20}; // means 10, 20, 0, 0, 0
```

Arrays anderer Basisdatentypen können auf dieselbe Weise initialisiert werden.

```
char arrayOfChars[5]; // declare the array and allocate the memory, don't initialize

char arrayOfChars[5] = { 'a', 'b', 'c', 'd', 'e' } ; //declare and initialize

double arrayOfDoubles[5] = {1.14159, 2.14159, 3.14159, 4.14159, 5.14159};

string arrayOfStrings[5] = { "C++", "is", "super", "duper", "great!"};
```

Beachten Sie auch, dass beim Zugriff auf Array-Elemente der Elementindex (oder die Position) des Arrays bei 0 beginnt.

```
int array[5] = { 10/*Element no.0*/, 20/*Element no.1*/, 30, 40, 50/*Element no.4*/};
std::cout << array[4]; //outputs 50
std::cout << array[0]; //outputs 10
```

Arrays online lesen: https://riptutorial.com/de/cplusplus/topic/3017/arrays

Kapitel 6: Art Abzug

Bemerkungen

Im November 2014 verabschiedete das C ++ Standardization Committee den Vorschlag N3922, der die spezielle Typabzugsregel für Auto- und Braced-Initialisierer mit direkter Initialisierungssyntax beseitigt. Dies ist nicht Teil des C ++ - Standards, wurde jedoch von einigen Compilern implementiert.

Examples

Vorlagenparameterabzug für Konstruktoren

Vor C ++ 17 kann ein Vorlagenabzug den Klassentyp für Sie nicht in einem Konstruktor ableiten. Es muss explizit angegeben werden. Manchmal können diese Typen jedoch sehr umständlich sein oder (im Fall von Lambdas) nicht benannt werden. `make_pair()` wir eine `make_pair()` `make_tuple()` wie `make_pair()` , `make_tuple()` , `back_inserter()` usw.) erhalten.

C ++ 17

Das ist nicht mehr nötig:

```
std::pair p(2, 4.5);     // std::pair<int, double>
std::tuple t(4, 3, 2.5); // std::tuple<int, int, double>
std::copy_n(vi1.begin(), 3,
    std::back_insert_iterator(vi2)); // constructs a back_insert_iterator<std::vector<int>>
std::lock_guard lk(mtx); // std::lock_guard<decltype(mtx)>
```

Es wird angenommen, dass Konstruktoren die Klassenvorlagenparameter ableiten. In einigen Fällen reicht dies jedoch nicht aus, und wir können explizite Abzugshilfen bereitstellen:

```
template <class Iter>
vector(Iter, Iter) -> vector<typename iterator_traits<Iter>::value_type>

int array[] = {1, 2, 3};
std::vector v(std::begin(array), std::end(array)); // deduces std::vector<int>
```

Vorlagentyp Abzug

Generische Vorlagen-Syntax

```
template<typename T>
void f(ParamType param);

f(expr);
```

Fall 1: `ParamType` ist eine Referenz oder ein Zeiger, jedoch keine Universal- oder

Weiterleitungsreferenz. In diesem Fall funktioniert der Typabzug auf diese Weise. Der Compiler ignoriert den Referenzteil, wenn er in `expr` . Der Compiler `expr` den Typ von `ParamType` mit `ParamType` und der Bestimmung von `T` `ParamType` .

```
template<typename T>
void f(T& param);       //param is a reference

int x = 27;             // x is an int
const int cx = x;       // cx is a const int
const int& rx = x;      // rx is a reference to x as a const int

f(x);                   // T is int, param's type is int&
f(cx);                  // T is const int, param's type is const int&
f(rx);                  // T is const int, param's type is const int&
```

Fall 2: `ParamType` ist eine Universalreferenz oder `ParamType` . In diesem Fall ist der `expr` derselbe wie in Fall 1, wenn der `expr` ein r- `expr` ist. Wenn `expr` ein lvalue ist, werden sowohl `T` als auch `ParamType` als lvalue-Referenzen abgeleitet.

```
template<typename T>
void f(T&& param);      // param is a universal reference

int x = 27;             // x is an int
const int cx = x;       // cx is a const int
const int& rx = x;      // rx is a reference to x as a const int

f(x);                   // x is lvalue, so T is int&, param's type is also int&
f(cx);                  // cx is lvalue, so T is const int&, param's type is also const int&
f(rx);                  // rx is lvalue, so T is const int&, param's type is also const int&
f(27);                  // 27 is rvalue, so T is int, param's type is therefore int&&
```

Fall 3: `ParamType` ist weder ein Zeiger noch eine Referenz. Wenn `expr` eine Referenz ist, wird der Referenzteil ignoriert. Wenn `expr` const ist, wird dies ebenfalls ignoriert. Wenn es flüchtig ist, wird es auch ignoriert, wenn der Typ von T abgeleitet wird.

```
template<typename T>
void f(T param);        // param is now passed by value

int x = 27;             // x is an int
const int cx = x;       // cx is a const int
const int& rx = x;      // rx is a reference to x as a const int

f(x);                   // T's and param's types are both int
f(cx);                  // T's and param's types are again both int
f(rx);                  // T's and param's types are still both int
```

Automatischer Typabzug

C ++ 11

Der Typabzug mit dem Schlüsselwort `auto` funktioniert fast genauso wie der Abzug des Vorlagentyps. Nachfolgend einige Beispiele:

```
auto x = 27;           // (x is neither a pointer nor a reference), x's type is int
const auto cx = x;     // (cx is neither a pointer nor a reference), cs's type is const int
const auto& rx = x;    // (rx is a non-universal reference), rx's type is a reference to a
const int

auto&& uref1 = x;      // x is int and lvalue, so uref1's type is int&
auto&& uref2 = cx;     // cx is const int and lvalue, so uref2's type is const int &
auto&& uref3 = 27;     // 27 is an int and rvalue, so uref3's type is int&&
```

Die Unterschiede sind nachfolgend aufgeführt:

```
auto x1 = 27;          // type is int, value is 27
auto x2(27);           // type is int, value is 27
auto x3 = { 27 };      // type is std::initializer_list<int>, value is { 27 }
auto x4{ 27 };         // type is std::initializer_list<int>, value is { 27 }
                       // in some compilers type may be deduced as an int with a
                       // value of 27. See remarks for more information.
auto x5 = { 1, 2.0 }   // error! can't deduce T for std::initializer_list<t>
```

Wie Sie sehen, wenn Sie geschweifte Initialisierer verwenden, wird auto dazu gezwungen, eine Variable vom Typ `std::initializer_list<T>` zu erstellen. Wenn der `T` , wird der Code abgelehnt.

Wenn `auto` als Rückgabetyp einer Funktion verwendet wird, wird angegeben, dass die Funktion einen nachgestellten Rückgabetyp hat .

```
auto f() -> int {
    return 42;
}
```

C ++ 14

C ++ 14 erlaubt zusätzlich zu den in C ++ 11 zugelassenen Verwendungen von auto Folgendes:

1. Bei Verwendung als Rückgabetyp einer Funktion ohne nachgestellten Rückgabetyp wird angegeben, dass der Rückgabetyp der Funktion aus den Rückgabeanweisungen im Rumpf der Funktion (sofern vorhanden) abgeleitet werden soll.

   ```
   // f returns int:
   auto f() { return 42; }
   // g returns void:
   auto g() { std::cout << "hello, world!\n"; }
   ```

2. Bei Verwendung im Parametertyp eines Lambda wird das Lambda als generisches Lambda definiert .

   ```
   auto triple = [](auto x) { return 3*x; };
   const auto x = triple(42); // x is a const int with value 126
   ```

Das spezielle Formular `decltype(auto)` leitet einen Typ unter Verwendung der `decltype` von `decltype` und nicht von `auto` .

```
int* p = new int(42);
```

```
auto x = *p;           // x has type int
decltype(auto) y = *p; // y is a reference to *p
```

In C ++ 03 und früheren Versionen hatte das Schlüsselwort `auto` eine völlig andere Bedeutung als ein von C geerbter Speicherklassenspezifizierer .

Art Abzug online lesen: https://riptutorial.com/de/cplusplus/topic/7863/art-abzug

Kapitel 7: Atomtypen

Syntax

- std :: atomic <T>
- std :: atomic_flag

Bemerkungen

`std::atomic` ermöglicht den atomaren Zugriff auf einen TriviallyCopyable-Typ. Es ist implementierungsabhängig, wenn dies über atomare Operationen oder Sperren erfolgt. Der einzige garantierte `std::atomic_flag` atomare Typ ist `std::atomic_flag` .

Examples

Multi-Threaded-Zugriff

Ein atomarer Typ kann verwendet werden, um sicher an einem Speicherplatz zu lesen und zu schreiben, der von zwei Threads gemeinsam genutzt wird.

Ein schlechtes Beispiel, das wahrscheinlich zu einem Datenrennen führt:

```
#include <thread>
#include <iostream>


//function will add all values including and between 'a' and 'b' to 'result'
void add(int a, int b, int * result) {
    for (int i = a; i <= b; i++) {
        *result += i;
    }
}

int main() {
    //a primitive data type has no thread safety
    int shared = 0;

    //create a thread that may run parallel to the 'main' thread
    //the thread will run the function 'add' defined above with paramters a = 1, b = 100,
result = &shared
    //analogous to 'add(1,100, &shared);'
    std::thread addingThread(add, 1, 100, &shared);

    //attempt to print the value of 'shared' to console
    //main will keep repeating this until the addingThread becomes joinable
    while (!addingThread.joinable()) {
        //this may cause undefined behavior or print a corrupted value
        //if the addingThread tries to write to 'shared' while the main thread is reading it
        std::cout << shared << std::endl;
    }
```

```
    //rejoin the thread at the end of execution for cleaning purposes
    addingThread.join();

    return 0;
}
```

Das obige Beispiel kann zu einem fehlerhaften Lesen führen und kann zu undefiniertem Verhalten führen.

Ein Beispiel mit Thread-Sicherheit:

```
#include <atomic>
#include <thread>
#include <iostream>


    //function will add all values including and between 'a' and 'b' to 'result'
void add(int a, int b, std::atomic<int> * result) {
    for (int i = a; i <= b; i++) {
        //atomically add 'i' to result
        result->fetch_add(i);
    }
}

int main() {
    //atomic template used to store non-atomic objects
    std::atomic<int> shared = 0;

    //create a thread that may run parallel to the 'main' thread
    //the thread will run the function 'add' defined above with paramters a = 1, b = 100,
result = &shared
    //analogous to 'add(1,100, &shared);'
    std::thread addingThread(add, 1, 10000, &shared);

    //print the value of 'shared' to console
    //main will keep repeating this until the addingThread becomes joinable
    while (!addingThread.joinable()) {
        //safe way to read the value of shared atomically for thread safe read
        std::cout << shared.load() << std::endl;
    }


    //rejoin the thread at the end of execution for cleaning purposes
    addingThread.join();

    return 0;
}
```

Das obige Beispiel ist sicher, da alle `store()` und `load()` Operationen des `atomic` Datentyps das gekapselte `int` vor dem gleichzeitigen Zugriff schützen.

Atomtypen online lesen: https://riptutorial.com/de/cplusplus/topic/3804/atomtypen

Kapitel 8: Attribute

Syntax

- [[Details]]: Einfaches Argument ohne Argument
- [[Details (Argumente)]]: Attribut mit Argumenten
- __attribute (Details): Nicht GCC / Clang / IBM-spezifisch
- __declspec (Details): Nicht-Standard MSVC-spezifisch

Examples

[[keine Rückkehr]]

C ++ 11

In C ++ 11 wurde das Attribut `[[noreturn]]` . Es kann für eine Funktion verwendet werden, um anzuzeigen, dass die Funktion nicht zum Aufrufer zurückkehrt, indem entweder eine *return*-Anweisung ausgeführt wird oder das Ende des Textes erreicht wird (es ist wichtig zu beachten, dass dies nicht für `void` Funktionen gilt, da sie dies sind) kehren Sie zum Anrufer zurück, sie geben nur keinen Wert zurück. Eine solche Funktion kann mit dem Aufruf von `std::terminate` oder `std::exit` oder mit dem Auslösen einer Ausnahme `std::terminate` . Es ist auch erwähnenswert, dass eine solche Funktion durch Ausführen von `longjmp` .

Die folgende Funktion `[[noreturn]]` zum Beispiel immer eine Ausnahme aus oder ruft `std::terminate` , daher ist sie ein guter Kandidat für `[[noreturn]]` :

```
[[noreturn]] void ownAssertFailureHandler(std::string message) {
    std::cerr << message << std::endl;
    if (THROW_EXCEPTION_ON_ASSERT)
        throw AssertException(std::move(message));
    std::terminate();
}
```

Durch diese Art von Funktionalität kann der Compiler eine Funktion ohne return-Anweisung beenden, wenn er weiß, dass der Code niemals ausgeführt wird. Da der Aufruf von `ownAssertFailureHandler` (oben definiert) im nachstehenden Code niemals zurückgegeben wird, muss der Compiler hier keinen Code unter diesem Aufruf hinzufügen:

```
std::vector<int> createSequence(int end) {
    if (end > 0) {
        std::vector<int> sequence;
        sequence.reserve(end+1);
        for (int i = 0; i <= end; ++i)
            sequence.push_back(i);
        return sequence;
```

```
    }
    ownAssertFailureHandler("Negative number passed to createSequence()"s);
    // return std::vector<int>{}; //< Not needed because of [[noreturn]]
}
```

Es ist undefiniertes Verhalten, wenn die Funktion tatsächlich zurückkehrt. Folgendes ist nicht zulässig:

```
[[noreturn]] void assertPositive(int number) {
    if (number >= 0)
        return;
    else
        ownAssertFailureHandler("Positive number expected"s); //< [[noreturn]]
}
```

Beachten Sie, dass der `[[noreturn]]` meist in void-Funktionen verwendet wird. Dies ist jedoch keine Voraussetzung, damit die Funktionen in der generischen Programmierung verwendet werden können:

```
template<class InconsistencyHandler>
double fortyTwoDivideBy(int i) {
    if (i == 0)
         i = InconsistencyHandler::correct(i);
    return 42. / i;
}

struct InconsistencyThrower {
    static [[noreturn]] int correct(int i) { ownAssertFailureHandler("Unknown
inconsistency"s); }
}

struct InconsistencyChangeToOne {
    static int correct(int i) { return 1; }
}

double fortyTwo = fortyTwoDivideBy<InconsistencyChangeToOne>(0);
double unreachable = fortyTwoDivideBy<InconsistencyThrower>(0);
```

Die folgenden Standardbibliotheksfunktionen haben dieses Attribut:

- std :: abbrechen
- std :: exit
- std :: quick_exit
- std :: unerwartet
- std :: terminate
- std :: rethrow_exception
- std :: throw_with_nested
- std :: nested_exception :: rethrow_nested

[[durchfallen]]

C ++ 17

Wenn ein `case` in einem `switch` , wird der Code des nächsten Falls ausgeführt. Letzteres kann mit der ´break'-Anweisung verhindert werden. Da dieses sogenannte Fallthrough-Verhalten zu Fehlern führen kann, wenn dies nicht beabsichtigt ist, warnen mehrere Compiler und statische Analysatoren dies.

Ab C ++ 17 wurde ein Standardattribut eingeführt, um anzuzeigen, dass die Warnung nicht erforderlich ist, wenn der Code durchfallen soll. Compiler können sicher `[[fallthrough]]` wenn ein Fall ohne `break` oder `[[fallthrough]]` und mindestens eine Anweisung `[[fallthrough]]` .

```
switch(input) {
    case 2011:
    case 2014:
    case 2017:
        std::cout << "Using modern C++" << std::endl;
        [[fallthrough]]; // > No warning
    case 1998:
    case 2003:
        standard = input;
}
```

In dem Vorschlag finden Sie detailliertere Beispiele, wie `[[fallthrough]]` verwendet werden kann.

[[veraltet]] und [[veraltet ("Grund")]]

C ++ 14

Mit C ++ 14 wurde eine Standardmethode eingeführt, um Funktionen über Attribute abzulehnen. `[[deprecated]]` kann verwendet werden, um anzuzeigen, dass eine Funktion veraltet ist. `[[deprecated("reason")]]` ermöglicht das Hinzufügen eines bestimmten Grunds, der vom Compiler angezeigt werden kann.

```
void function(std::unique_ptr<A> &&a);

// Provides specific message which helps other programmers fixing there code
[[deprecated("Use the variant with unique_ptr instead, this function will be removed in the
next release")]]
void function(std::auto_ptr<A> a);

// No message, will result in generic warning if called.
[[deprecated]]
void function(A *a);
```

Dieses Attribut kann angewendet werden auf:

- die Deklaration einer Klasse
- ein typedef-name
- eine Variable
- ein nicht statisches Datenelement
- eine Funktion
- eine Aufzählung
- eine Vorlagenspezialisierung

(Ref. C ++ 14 Standardentwurf : 7.6.5 Veraltetes Attribut)

[nodiscard]]

C ++ 17

Das Attribut `[[nodiscard]]` kann verwendet werden, um anzuzeigen, dass der Rückgabewert einer Funktion beim Funktionsaufruf nicht ignoriert werden soll. Wenn der Rückgabewert ignoriert wird, sollte der Compiler eine Warnung ausgeben. Das Attribut kann hinzugefügt werden:

- Eine Funktionsdefinition
- Eine Art

Das Hinzufügen des Attributs zu einem Typ hat dasselbe Verhalten wie das Hinzufügen des Attributs zu jeder einzelnen Funktion, die diesen Typ zurückgibt.

```
template<typename Function>
[[nodiscard]] Finally<std::decay_t<Function>> onExit(Function &&f);

void f(int &i) {
    assert(i == 0);                     // Just to make comments clear!
    ++i;                                // i == 1
    auto exit1 = onExit([&i]{ --i; }); // Reduce by 1 on exiting f()
    ++i;                                // i == 2
    onExit([&i]{ --i; });               // BUG: Reducing by 1 directly
                                        //      Compiler warning expected
    std::cout << i << std::end;         // Expected: 2, Real: 1
}
```

Siehe den Vorschlag für detailliertere Beispiele, wie `[[nodiscard]]` verwendet werden kann.

Hinweis: Die Details der Implementierung `Finally` / `onExit` sind in dem Beispiel weggelassen, siehe Schließlich / ScopeExit .

[[vielleicht_unused]]

Das Attribut `[[maybe_unused]]` wird erstellt, um im Code `[[maybe_unused]]` dass bestimmte Logik möglicherweise nicht verwendet wird. Dies wird häufig mit Präprozessor-Bedingungen verknüpft, bei denen dies verwendet werden kann oder nicht verwendet werden kann. Da Compiler zu nicht verwendeten Variablen Warnungen ausgeben können, können sie auf diese Weise durch Angabe von Absichten unterdrückt werden.

Ein typisches Beispiel für Variablen, die in Debugbuilds benötigt werden, während sie in der Produktion nicht benötigt werden, sind Rückgabewerte, die den Erfolg anzeigen. In den Debug-Builds sollte die Bedingung geltend gemacht werden, obwohl diese Assays in der Produktion entfernt wurden.

```
[[maybe_unused]] auto mapInsertResult = configuration.emplace("LicenseInfo",
stringifiedLicenseInfo);
assert(mapInsertResult.second); // We only get called during startup, so we can't be in the
map
```

Ein komplexeres Beispiel sind verschiedene Arten von Hilfsfunktionen, die sich in einem unbenannten Namensraum befinden. Wenn diese Funktionen beim Kompilieren nicht verwendet werden, gibt der Compiler möglicherweise eine Warnung aus. Im Idealfall möchten Sie sie mit den gleichen Präprozessor-Tags wie der Aufrufer `[[maybe_unused]]` dies jedoch komplex werden kann, ist das Attribut `[[maybe_unused]]` eine eher wartbare Alternative.

```
namespace {
    [[maybe_unused]] std::string createWindowsConfigFilePath(const std::string &relativePath);
    // TODO: Reuse this on BSD, MAC ...
    [[maybe_unused]] std::string createLinuxConfigFilePath(const std::string &relativePath);
}

std::string createConfigFilePath(const std::string &relativePath) {
#if OS == "WINDOWS"
      return createWindowsConfigFilePath(relativePath);
#elif OS == "LINUX"
      return createLinuxConfigFilePath(relativePath);
#else
#error "OS is not yet supported"
#endif
}
```

In dem Vorschlag finden Sie detailliertere Beispiele zur Verwendung von `[[maybe_unused]]` .

Attribute online lesen: https://riptutorial.com/de/cplusplus/topic/5251/attribute

Kapitel 9: Aufzählung

Examples

Grundlegende Aufzählungserklärung

Bei Standard-Enumerationen können Benutzer einen nützlichen Namen für eine Menge von Ganzzahlen angeben. Die Namen werden zusammen als Enumeratoren bezeichnet. Eine Aufzählung und die zugehörigen Enumeratoren sind wie folgt definiert:

```
enum myEnum
{
    enumName1,
    enumName2,
};
```

Eine Aufzählung ist ein *Typ* , der sich von allen anderen Typen unterscheidet. In diesem Fall lautet der Name dieses Typs `myEnum` . Es wird erwartet, dass Objekte dieses Typs den Wert eines Enumerators innerhalb der Enumeration annehmen.

Die in der Aufzählung deklarierten Enumeratoren sind konstante Werte des Aufzählungstyps. Obwohl die Enumeratoren innerhalb des Typs deklariert sind, ist der Bereichsoperator `::` für den Zugriff auf den Namen nicht erforderlich. Der Name des ersten Enumerators lautet also `enumName1` .

C ++ 11

Mit dem Bereichsoperator kann optional auf einen Enumerator innerhalb einer Enumeration zugegriffen werden. `enumName1` kann also auch als `myEnum::enumName1` .

Enumeratoren werden ganzzahlige Werte zugewiesen, beginnend mit 0 und für jeden Enumerator in einer Enumeration um 1 erhöht. Im obigen Fall hat `enumName1` den Wert 0, während `enumName2` den Wert 1 hat.

Enumeratoren können vom Benutzer auch mit einem bestimmten Wert belegt werden. Dieser Wert muss ein integraler konstanter Ausdruck sein. Enumeratoren, deren Werte nicht explizit angegeben werden, werden auf den Wert des vorherigen Enumerators + 1 gesetzt.

```
enum myEnum
{
    enumName1 = 1, // value will be 1
    enumName2 = 2, // value will be 2
    enumName3,     // value will be 3, previous value + 1
    enumName4 = 7, // value will be 7
    enumName5,     // value will be 8
    enumName6 = 5, // value will be 5, legal to go backwards
    enumName7 = 3, // value will be 3, legal to reuse numbers
    enumName8 = enumName4 + 2, // value will be 9, legal to take prior enums and adjust them
};
```

Aufzählung in switch-Anweisungen

Enumeratoren werden häufig für switch-Anweisungen verwendet und erscheinen daher häufig in Zustandsmaschinen. Tatsächlich ist eine nützliche Funktion von switch-Anweisungen mit Enumerationen, dass der Compiler eine Warnung ausgibt, wenn keine Standardanweisung für den Switch enthalten ist und nicht alle Werte der Enumeration verwendet wurden.

```
enum State {
    start,
    middle,
    end
};

...

switch(myState) {
    case start:
       ...
    case middle:
       ...
} // warning: enumeration value 'end' not handled in switch [-Wswitch]
```

Iteration über eine Aufzählung

Es ist kein integriertes Programm vorhanden, um die Aufzählung zu durchlaufen.

Es gibt jedoch mehrere Möglichkeiten

- für eine `enum` mit nur aufeinander folgenden Werten:

  ```
  enum E {
      Begin,
      E1 = Begin,
      E2,
      // ..
      En,
      End
  };

  for (E e = E::Begin; e != E::End; ++e) {
      // Do job with e
  }
  ```

C ++ 11

Mit der `enum class` muss `operator ++` implementiert werden:

```
E& operator ++ (E& e)
{
    if (e == E::End) {
        throw std::out_of_range("for E& operator ++ (E&)");
    }
    e = E(static_cast<std::underlying_type<E>::type>(e) + 1);
    return e;
```

```
}
```

- Verwenden eines Containers als `std::vector`

```
enum E {
    E1 = 4,
    E2 = 8,
    // ..
    En
};

std::vector<E> build_all_E()
{
    const E all[] = {E1, E2, /*..*/ En};

    return std::vector<E>(all, all + sizeof(all) / sizeof(E));
}

std::vector<E> all_E = build_all_E();
```

und dann

```
for (std::vector<E>::const_iterator it = all_E.begin(); it != all_E.end(); ++it) {
    E e = *it;
    // Do job with e;
}
```

C ++ 11

- oder `std::initializer_list` und eine einfachere Syntax:

```
enum E {
    E1 = 4,
    E2 = 8,
    // ..
    En
};

constexpr std::initializer_list<E> all_E = {E1, E2, /*..*/ En};
```

und dann

```
for (auto e : all_E) {
    // Do job with e
}
```

Umfangreiche Aufzählungen

In C ++ 11 werden so *genannte Bereichsumgebungen eingeführt* . Dies sind Aufzählungen, deren Mitglieder mit `enumname::membername` qualifiziert werden `enumname::membername` . Bereichs-Enums werden mit der `enum class` deklariert. So speichern Sie beispielsweise die Farben in einem Regenbogen:

```
enum class rainbow {
    RED,
    ORANGE,
    YELLOW,
    GREEN,
    BLUE,
    INDIGO,
    VIOLET
};
```

Um auf eine bestimmte Farbe zuzugreifen:

```
rainbow r = rainbow::INDIGO;
```

`enum class` es können nicht ohne Besetzung implizit in `int` s konvertiert werden. Also ist `int x = rainbow::RED` ungültig.

Mit Aufzählungsumgebungen können Sie auch den *zugrunde liegenden Typ* angeben, dh den *Typ* , der zur Darstellung eines Members verwendet wird. Standardmäßig ist es `int` . In einem Tic-Tac-Toe-Spiel können Sie das Stück als speichern

```
enum class piece : char {
    EMPTY = '\0',
    X = 'X',
    O = 'O',
};
```

Wie Sie feststellen können, `enum` kann s ein nachlauf Komma nach dem letzten Mitglied haben.

Vorwärtsdeklaration in C ++ 11 auflisten

Umfangreiche Aufzählungen:

```
...
enum class Status; // Forward declaration
Status doWork(); // Use the forward declaration
...
enum class Status { Invalid, Success, Fail };
Status doWork() // Full declaration required for implementation
{
    return Status::Success;
}
```

Nicht begrenzte Aufzählungen:

```
...
enum Status: int; // Forward declaration, explicit type required
Status doWork(); // Use the forward declaration
...
enum Status: int{ Invalid=0, Success, Fail }; // Must match forward declare type
static_assert( Success == 1 );
```

Ein ausführliches Beispiel für mehrere Dateien finden Sie hier: Beispiel für blinde Obsthändler

Aufzählung online lesen: https://riptutorial.com/de/cplusplus/topic/2796/aufzahlung

Kapitel 10: Ausdrücke falten

Bemerkungen

Faltenausdrücke werden für die folgenden Operatoren unterstützt

+	-	*	/	%	\ ^	&	|	<<	>>		
+ =	- =	* =	/ =	% =	\ ^ =	& =	| =	<< =	>> =	=	
==	! =	<	>	<=	> =	&&	||	.	. *	-> *	

Beim Falten einer leeren Sequenz wird ein Falzausdruck mit Ausnahme der folgenden drei Operatoren falsch gebildet:

Operator	Wert, wenn das Parameterpaket leer ist
&&	wahr
||	falsch
.	Leere()

Examples

Unäre Falten

Unäre Falten werden verwendet, *um* Parameterpakete über einen bestimmten Operator zu *falten* . Es gibt zwei Arten von unären Falten:

- Unary **Left** Fold `(... op pack)` das wie folgt erweitert wird:

```
((Pack1 op Pack2) op ...) op PackN
```

- Unary **Right** Fold `(pack op ...)` das wie folgt erweitert wird:

```
Pack1 op (... (Pack(N-1) op PackN))
```

Hier ist ein Beispiel

```
template<typename... Ts>
int sum(Ts... args)
{
    return (... + args); //Unary left fold
    //return (args + ...); //Unary right fold
```

```
    // The two are equivalent if the operator is associative.
    // For +, ((1+2)+3) (left fold) == (1+(2+3)) (right fold)
    // For -, ((1-2)-3) (left fold) != (1-(2-3)) (right fold)
}

int result = sum(1, 2, 3); //  6
```

Binäre Falten

Binäre Falten sind im Wesentlichen unäre Falten mit einem zusätzlichen Argument.

Es gibt zwei Arten von Binärfalten:

- Binary **Left** Fold - `(value op ... op pack)` - Erweitert sich wie folgt:

  ```
  (((Value op Pack1) op Pack2) op ...) op PackN
  ```

- Binary **Right** Fold `(pack op ... op value)` - Erweitert wie folgt:

  ```
  Pack1 op (... op (Pack(N-1) op (PackN op Value)))
  ```

Hier ist ein Beispiel:

```
template<typename... Ts>
int removeFrom(int num, Ts... args)
{
    return (num - ... - args); //Binary left fold
    // Note that a binary right fold cannot be used
    // due to the lack of associativity of operator-
}

int result = removeFrom(1000, 5, 10, 15); //'result' is 1000 - 5 - 10 - 15 = 970
```

Ein Komma falten

Es ist eine übliche Operation, dass eine bestimmte Funktion für jedes Element in einem Parameterpaket ausgeführt werden muss. Mit C ++ 11 ist das Beste, was wir tun können:

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    using expander = int[];
    (void)expander{0,
        (void(os << args), 0)...
    };
}
```

Mit einem Falzausdruck vereinfacht sich das Obige jedoch:

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    (void(os << args), ...);
```

```
}
```

Keine kryptische Heizplatte erforderlich.

Ausdrücke falten online lesen: https://riptutorial.com/de/cplusplus/topic/2676/ausdrucke-falten

Kapitel 11: Ausdrucksvorlagen

Examples

Grundlegende Ausdrucksvorlagen für elementweise algebraische Ausdrücke

Einführung und Motivation

Ausdrucksvorlagen (im Folgenden als **ETs** bezeichnet) sind eine leistungsfähige Vorlagenmeta-Programmiertechnik, mit der die Berechnung von manchmal recht teuren Ausdrücken beschleunigt wird. Es wird häufig in verschiedenen Bereichen verwendet, beispielsweise bei der Implementierung von linearen Algebra-Bibliotheken.

Betrachten Sie für dieses Beispiel den Kontext der linearen algebraischen Berechnungen. Insbesondere Berechnungen, die nur **elementweise Operationen umfassen** . Diese Art von Berechnungen sind die grundlegendsten Anwendungen von **ETs** und sie sind eine gute Einführung in die interne Arbeitsweise von **ETs** .

Schauen wir uns ein motivierendes Beispiel an. Betrachten Sie die Berechnung des Ausdrucks:

```
Vector vec_1, vec_2, vec_3;

// Initializing vec_1, vec_2 and vec_3.

Vector result = vec_1 + vec_2*vec_3;
```

Der Einfachheit halber gehe ich davon aus, dass die Klassen `Vector` und operation + (vector plus: elementweise plus Operation) und operation * (hier bedeutet vector inneres Produkt: auch elementweise Operation) beide korrekt implementiert sind, wie z wie sie mathematisch sein sollten.

Bei einer herkömmlichen Implementierung ohne Verwendung von **ETs** (oder anderen ähnlichen Techniken) finden **mindestens fünf** Konstruktionen von `Vector` Instanzen statt, um das `result` :

1. Drei Instanzen, die `vec_1` , `vec_2` und `vec_3` .
2. Eine temporäre `Vector` Instanz `_tmp` , die das Ergebnis von `_tmp = vec_2*vec_3;` .
3. Bei richtiger Verwendung der **Rückgabewertoptimierung schließlich** wird die Erstellung des `result` in `result = vec_1 + _tmp;` .

Die Implementierung mit **ETs** kann **die Erstellung von temporärem** `Vector _tmp` in 2 verhindern, sodass nur **vier** Konstruktionen von `Vector` Instanzen übrig `Vector _tmp` . Interessanter ist der folgende Ausdruck, der komplexer ist:

```
Vector result = vec_1 + (vec_2*vec3 + vec_1)*(vec_2 + vec_3*vec_1);
```

Insgesamt gibt es vier Konstruktionen von `Vector` Instanzen: `vec_1, vec_2, vec_3` und `result` . Mit anderen Worten, in diesem Beispiel, **bei dem nur elementweise Operationen beteiligt sind** , ist gewährleistet, dass **aus Zwischenberechnungen keine temporären Objekte erstellt werden** .

Wie funktionieren ETs?

Grundsätzlich bestehen **ETs für algebraische Berechnungen** aus zwei Bausteinen:

1. **Reine algebraische Ausdrücke** (**PAE**): Sie sind Proxies / Abstraktionen algebraischer Ausdrücke. Eine reine Algebraik führt keine tatsächlichen Berechnungen durch, sondern ist lediglich eine Abstraktion / Modellierung des Berechnungs-Workflows. Eine PAE kann ein Modell **entweder der Eingabe oder der Ausgabe von algebraischen Ausdrücken sein** . Instanzen von **PAEs** werden oft als billig betrachtet.
2. **Faule Auswertungen** : Dies sind Implementierungen realer Berechnungen. Im folgenden Beispiel werden wir sehen, dass für Ausdrücke, die nur elementweise Operationen enthalten, Lazy-Auswertungen tatsächliche Berechnungen innerhalb der indizierten Zugriffsoperation für das Endergebnis ausführen können, wodurch ein Bewertungsschema nach Bedarf erstellt wird: Eine Berechnung wird nicht durchgeführt Nur wenn auf das Endergebnis zugegriffen wird.

Also, wie implementieren wir in diesem Beispiel genau **ETs** ? Lass uns jetzt durchgehen.

Beachten Sie immer den folgenden Code-Ausschnitt:

```
Vector vec_1, vec_2, vec_3;

// Initializing vec_1, vec_2 and vec_3.

Vector result = vec_1 + vec_2*vec_3;
```

Der zu berechnende Ausdruck kann weiter in zwei Unterausdrücke zerlegt werden:

1. Ein Vektor plus Ausdruck (als **plus_expr bezeichnet**)
2. Ein Vektor-Produktausdruck (bezeichnet als **innerprod_expr**).

Was machen die **ETs** ?

- Statt jeden Unterausdruck sofort zu berechnen, modellieren die **ETs** zunächst den gesamten Ausdruck anhand einer grafischen Struktur. Jeder Knoten in der Grafik repräsentiert eine **PAE** . Die Kantenverbindung der Knoten repräsentiert den tatsächlichen Rechenfluss. Für den obigen Ausdruck erhalten wir die folgende Grafik:

```
       result = plus_expr( vec_1, innerprod_expr(vec_2, vec_3) )
          /   \
         /     \
        /       \
       /   innerprod_expr( vec_2, vec_3 )
      /         /  \
     /         /    \
```

```
    /         /      \
 vec_1     vec_2    vec_3
```

- Die abschließende Berechnung wird durch einen **Blick in die Diagrammhierarchie** implementiert: Da es sich hier **nur um elementweise** Operationen handelt, kann die Berechnung jedes indizierten Werts im `result` **unabhängig voneinander erfolgen** : Die abschließende Auswertung des `result` kann **träge** auf ein **Element** verschoben **werden. weise Bewertung** jedes `result` . Mit anderen Worten, da die Berechnung eines Elements von `result` , `elem_res` , unter Verwendung entsprechender Elemente in `vec_1` (`elem_1`), `vec_2` (`elem_2`) und `vec_3` (`elem_3`) `elem_3` werden kann:

```
elem_res = elem_1 + elem_2*elem_3;
```

Es ist daher nicht erforderlich, einen temporären `Vector` zu erstellen, um das Ergebnis des Zwischenprodukts zu speichern: **Die gesamte Berechnung für ein Element kann vollständig ausgeführt und innerhalb der indizierten Zugriffsoperation codiert werden** .

Hier sind die Beispielcodes in Aktion.

Datei vec.hh: Wrapper für std :: vector, um Protokoll anzuzeigen, wenn eine Konstruktion aufgerufen wird.

```
#ifndef EXPR_VEC
# define EXPR_VEC

# include <vector>
# include <cassert>
# include <utility>
# include <iostream>
# include <algorithm>
# include <functional>

///
/// This is a wrapper for std::vector. It's only purpose is to print out a log when a
/// vector constructions in called.
/// It wraps the indexed access operator [] and the size() method, which are
/// important for later ETs implementation.
///

// std::vector wrapper.
template<typename ScalarType> class Vector
{
public:
  explicit Vector() { std::cout << "ctor called.\n"; };
  explicit Vector(int size): _vec(size) { std::cout << "ctor called.\n"; };
  explicit Vector(const std::vector<ScalarType> &vec): _vec(vec)
  { std::cout << "ctor called.\n"; };

  Vector(const Vector<ScalarType> & vec): _vec{vec()}
  { std::cout << "copy ctor called.\n"; };
```

```
  Vector(Vector<ScalarType> && vec): _vec(std::move(vec()))
  { std::cout << "move ctor called.\n"; };

  Vector<ScalarType> & operator=(const Vector<ScalarType> &) = default;
  Vector<ScalarType> & operator=(Vector<ScalarType> &&) = default;

  decltype(auto) operator[](int indx) { return _vec[indx]; }
  decltype(auto) operator[](int indx) const { return _vec[indx]; }

  decltype(auto) operator()() & { return (_vec); };
  decltype(auto) operator()() const & { return (_vec); };
  Vector<ScalarType> && operator()() && { return std::move(*this); }

  int size() const { return _vec.size(); }

private:
  std::vector<ScalarType> _vec;
};

///
/// These are conventional overloads of operator + (the vector plus operation)
/// and operator * (the vector inner product operation) without using the expression
/// templates. They are later used for bench-marking purpose.
///

// + (vector plus) operator.
template<typename ScalarType>
auto operator+(const Vector<ScalarType> &lhs, const Vector<ScalarType> &rhs)
{
  assert(lhs().size() == rhs().size() &&
         "error: ops plus -> lhs and rhs size mismatch.");

  std::vector<ScalarType> _vec;
  _vec.resize(lhs().size());
  std::transform(std::cbegin(lhs()), std::cend(lhs()),
                 std::cbegin(rhs()), std::begin(_vec),
                 std::plus<>());
  return Vector<ScalarType>(std::move(_vec));
}

// * (vector inner product) operator.
template<typename ScalarType>
auto operator*(const Vector<ScalarType> &lhs, const Vector<ScalarType> &rhs)
{
  assert(lhs().size() == rhs().size() &&
         "error: ops multiplies -> lhs and rhs size mismatch.");

  std::vector<ScalarType> _vec;
  _vec.resize(lhs().size());
  std::transform(std::cbegin(lhs()), std::cend(lhs()),
                 std::cbegin(rhs()), std::begin(_vec),
                 std::multiplies<>());
  return Vector<ScalarType>(std::move(_vec));
}

#endif //!EXPR_VEC
```

Datei expr.hh: Implementierung von Ausdrucksvorlagen für

elementweise Operationen (vector plus und vector inner product)

Lassen Sie uns es in Abschnitte aufteilen.

1. Abschnitt 1 implementiert eine Basisklasse für alle Ausdrücke. Es verwendet das Curiously **Recurring Template Pattern** (CRTP).
2. Abschnitt 2 implementiert die erste **PAE** : ein **Terminal** , das nur ein Wrapper (const-Referenz) einer Eingabedatenstruktur ist, die einen realen Eingabewert für die Berechnung enthält.
3. Abschnitt 3 implementiert die zweite **PAE** : **binary_operation** , eine Klassenvorlage, die später für vector_plus und vector_innerprod verwendet wird. Sie wird durch die **Art der Bedienung** , **die linke PAE** und **die rechte PAE** parametrisiert. Die tatsächliche Berechnung wird im Operator für indizierten Zugriff codiert.
4. Abschnitt 4 definiert die Operationen vector_plus und vector_innerprod als **elementweise Operationen** . Außerdem werden die Operatoren + und * für **PAEs** überladen, sodass diese beiden Operationen auch **PAE zurückgeben** .

```
#ifndef EXPR_EXPR
# define EXPR_EXPR


/// Fwd declaration.
template<typename> class Vector;

namespace expr
{


/// ----------------------------------------
///
/// Section 1.
///
/// The first section is a base class template for all kinds of expression. It
/// employs the Curiously Recurring Template Pattern, which enables its instantiation
/// to any kind of expression structure inheriting from it.
///
/// ----------------------------------------


  /// Base class for all expressions.
  template<typename Expr> class expr_base
  {
  public:
    const Expr& self() const { return static_cast<const Expr&>(*this); }
    Expr& self() { return static_cast<Expr&>(*this); }

  protected:
    explicit expr_base() {};
    int size() const { return self().size_impl(); }
    auto operator[](int indx) const { return self().at_impl(indx); }
    auto operator()() const { return self()(); };
  };
```

```
/// ----------------------------------------
///
/// The following section 2 & 3 are abstractions of pure algebraic expressions (PAE).
/// Any PAE can be converted to a real object instance using operator(): it is in
/// this conversion process, where the real computations are done.

///
/// Section 2. Terminal
///
/// A terminal is an abstraction wrapping a const reference to the Vector data
/// structure. It inherits from expr_base, therefore providing a unified interface
/// wrapping a Vector into a PAE.
///
/// It provides the size() method, indexed access through at_impl() and a conversion
/// to referenced object through () operator.
///
/// It might no be necessary for user defined data structures to have a terminal
/// wrapper, since user defined structure can inherit expr_base, therefore eliminates
/// the need to provide such terminal wrapper.
///
/// ----------------------------------------


  /// Generic wrapper for underlying data structure.
  template<typename DataType> class terminal: expr_base<terminal<DataType>>
  {
  public:
    using base_type = expr_base<terminal<DataType>>;
    using base_type::size;
    using base_type::operator[];
    friend base_type;

    explicit terminal(const DataType &val): _val(val) {}
    int size_impl() const { return _val.size(); };
    auto at_impl(int indx) const { return _val[indx]; };
    decltype(auto) operator()() const { return (_val); }

  private:
    const DataType &_val;
  };


/// ----------------------------------------
///
/// Section 3. Binary operation expression.
///
/// This is a PAE abstraction of any binary expression. Similarly it inherits from
/// expr_base.
///
/// It provides the size() method, indexed access through at_impl() and a conversion
/// to referenced object through () operator. Each call to the at_impl() method is
/// a element wise computation.
///
/// ----------------------------------------


  /// Generic wrapper for binary operations (that are element-wise).
  template<typename Ops, typename lExpr, typename rExpr>
  class binary_ops: public expr_base<binary_ops<Ops,lExpr,rExpr>>
```

```
  {
  public:
    using base_type = expr_base<binary_ops<Ops,lExpr,rExpr>>;
    using base_type::size;
    using base_type::operator[];
    friend base_type;

    explicit binary_ops(const Ops &ops, const lExpr &lxpr, const rExpr &rxpr)
      : _ops(ops), _lxpr(lxpr), _rxpr(rxpr) {};
    int size_impl() const { return _lxpr.size(); };

    /// This does the element-wise computation for index indx.
    auto at_impl(int indx) const { return _ops(_lxpr[indx], _rxpr[indx]); };

    /// Conversion from arbitrary expr to concrete data type. It evaluates
    /// element-wise computations for all indices.
    template<typename DataType> operator DataType()
    {
      DataType _vec(size());
      for(int _ind = 0; _ind < _vec.size(); ++_ind)
        _vec[_ind] = (*this)[_ind];
      return _vec;
    }

  private: /// Ops and expr are assumed cheap to copy.
    Ops   _ops;
    lExpr _lxpr;
    rExpr _rxpr;
  };


/// ----------------------------------------
/// Section 4.
///
/// The following two structs defines algebraic operations on PAEs: here only vector
/// plus and vector inner product are implemented.
///
/// First, some element-wise operations are defined : in other words, vec_plus and
/// vec_prod acts on elements in Vectors, but not whole Vectors.
///
/// Then, operator + & * are overloaded on PAEs, such that: + & * operations on PAEs
/// also return PAEs.
///
/// ----------------------------------------


  /// Element-wise plus operation.
  struct vec_plus_t
  {
    constexpr explicit vec_plus_t() = default;
    template<typename LType, typename RType>
    auto operator()(const LType &lhs, const RType &rhs) const
    { return lhs+rhs; }
  };

  /// Element-wise inner product operation.
  struct vec_prod_t
  {
    constexpr explicit vec_prod_t() = default;
    template<typename LType, typename RType>
    auto operator()(const LType &lhs, const RType &rhs) const
```

```
    { return lhs*rhs; }
  };

  /// Constant plus and inner product operator objects.
  constexpr vec_plus_t vec_plus{};
  constexpr vec_prod_t vec_prod{};

  /// Plus operator overload on expressions: return binary expression.
  template<typename lExpr, typename rExpr>
  auto operator+(const lExpr &lhs, const rExpr &rhs)
  { return binary_ops<vec_plus_t,lExpr,rExpr>(vec_plus,lhs,rhs); }

  /// Inner prod operator overload on expressions: return binary expression.
  template<typename lExpr, typename rExpr>
  auto operator*(const lExpr &lhs, const rExpr &rhs)
  { return binary_ops<vec_prod_t,lExpr,rExpr>(vec_prod,lhs,rhs); }

} //!expr


#endif //!EXPR_EXPR
```

Datei main.cc: Test-SRC-Datei

```
# include <chrono>
# include <iomanip>
# include <iostream>
# include "vec.hh"
# include "expr.hh"
# include "boost/core/demangle.hpp"


int main()
{
  using dtype = float;
  constexpr int size = 5e7;

  std::vector<dtype> _vec1(size);
  std::vector<dtype> _vec2(size);
  std::vector<dtype> _vec3(size);

  // ... Initialize vectors' contents.

  Vector<dtype> vec1(std::move(_vec1));
  Vector<dtype> vec2(std::move(_vec2));
  Vector<dtype> vec3(std::move(_vec3));

  unsigned long start_ms_no_ets =
    std::chrono::duration_cast<std::chrono::milliseconds>
    (std::chrono::system_clock::now().time_since_epoch()).count();
  std::cout << "\nNo-ETs evaluation starts.\n";

  Vector<dtype> result_no_ets = vec1 + (vec2*vec3);

  unsigned long stop_ms_no_ets =
    std::chrono::duration_cast<std::chrono::milliseconds>
    (std::chrono::system_clock::now().time_since_epoch()).count();
```

```
  std::cout << std::setprecision(6) << std::fixed
            << "No-ETs. Time eclapses: " << (stop_ms_no_ets-start_ms_no_ets)/1000.0
            << " s.\n" << std::endl;

  unsigned long start_ms_ets =
    std::chrono::duration_cast<std::chrono::milliseconds>
    (std::chrono::system_clock::now().time_since_epoch()).count();
  std::cout << "Evaluation using ETs starts.\n";

  expr::terminal<Vector<dtype>> vec4(vec1);
  expr::terminal<Vector<dtype>> vec5(vec2);
  expr::terminal<Vector<dtype>> vec6(vec3);

  Vector<dtype> result_ets = (vec4 + vec5*vec6);

  unsigned long stop_ms_ets =
    std::chrono::duration_cast<std::chrono::milliseconds>
    (std::chrono::system_clock::now().time_since_epoch()).count();
  std::cout << std::setprecision(6) << std::fixed
            << "With ETs. Time eclapses: " << (stop_ms_ets-start_ms_ets)/1000.0
            << " s.\n" << std::endl;

  auto ets_ret_type = (vec4 + vec5*vec6);
  std::cout << "\nETs result's type:\n";
  std::cout << boost::core::demangle( typeid(decltype(ets_ret_type)).name() ) << '\n';

  return 0;
}
```

Beim Kompilieren mit `-O3 -std=c++14` Verwendung von GCC 5.3 ist eine mögliche Ausgabe möglich:

```
ctor called.
ctor called.
ctor called.

No-ETs evaluation starts.
ctor called.
ctor called.
No-ETs. Time eclapses: 0.571000 s.

Evaluation using ETs starts.
ctor called.
With ETs. Time eclapses: 0.164000 s.


ETs result's type:
expr::binary_ops<expr::vec_plus_t, expr::terminal<Vector<float> >,
expr::binary_ops<expr::vec_prod_t, expr::terminal<Vector<float> >,
expr::terminal<Vector<float> > > >
```

Die Beobachtungen sind:

- Die Verwendung von **ETs** erzielt **in diesem Fall** eine ziemlich deutliche Leistungssteigerung (> 3x).
- Die Erstellung eines temporären Vector-Objekts wird eliminiert. Wie im Fall der **ET** wird ctor nur einmal aufgerufen.

- Boost :: demangle wurde verwendet, um die Art der Rückgabe von ETs vor der Konvertierung zu visualisieren: Es konstruierte genau den gleichen Ausdruck wie oben gezeigt.

Nachteile und Vorbehalte

- Ein offensichtlicher Nachteil von **ETs** ist die Lernkurve, die Komplexität der Implementierung und der Schwierigkeitsgrad der **Codewartung** . In dem obigen Beispiel, in dem nur elementweise Operationen betrachtet werden, enthält die Implementierung bereits eine enorme Anzahl von Boilerplates, geschweige denn in der realen Welt, wo komplexere algebraische Ausdrücke in jeder Berechnung vorkommen und die Unabhängigkeit von Elementen nicht mehr zutrifft (zum Beispiel Matrixmultiplikation)) wird der Schwierigkeitsgrad exponentiell sein.

- Ein weiterer Nachteil bei der Verwendung von **ETs** ist, dass sie mit dem Schlüsselwort `auto` gut funktionieren. Wie oben erwähnt, handelt es sich bei **PAEs** im Wesentlichen um Proxies: Proxies spielen im Grunde nicht gut mit `auto` . Betrachten Sie das folgende Beispiel:

```
auto result = ...;                // Some expensive expression:
                                  // auto returns the expr graph,
                                  // NOT the computed value.
for(auto i = 0; i < 100; ++i)
    ScalrType value = result* ... // Some other expensive computations using result.
```

Hier **wird das Ergebnis in jeder Iteration der for-Schleife erneut ausgewertet** , da anstelle des berechneten Werts der Ausdrucksgraph an die for-Schleife übergeben wird.

Bestehende Bibliotheken, die **ETs** implementieren

- **boost :: proto** ist eine leistungsstarke Bibliothek, in der Sie Ihre eigenen Regeln und Grammatiken für Ihre eigenen Ausdrücke definieren und mit **ETs** ausführen können.
- **Eigen** ist eine Bibliothek für lineare Algebra, die verschiedene algebraische Berechnungen effizient mit **ETs** implementiert.

Ein einfaches Beispiel, das Ausdrucksvorlagen veranschaulicht

Eine Ausdrucksvorlage ist eine Kompilierungszeitoptimierungstechnik, die hauptsächlich im wissenschaftlichen Rechnen verwendet wird. Ihr Hauptzweck besteht darin, unnötige temporäre Werte zu vermeiden und die Schleifenberechnungen mit einem einzigen Durchgang zu optimieren (normalerweise, wenn Operationen mit numerischen Aggregaten ausgeführt werden). Anfangs wurden Ausdrucksvorlagen entwickelt, um die Ineffizienz der naiven Überladung von Operatoren bei der Implementierung numerischer `Array` oder `Matrix` Typen zu umgehen. Eine äquivalente Terminologie für Ausdrucksvorlagen wurde von Bjarne Stroustrup eingeführt, der sie in der neuesten Version seines Buches "Die C ++ - Programmiersprache" als "fusionierte Operationen" bezeichnet.

Bevor Sie tatsächlich in Ausdrucksvorlagen eintauchen, sollten Sie verstehen, warum Sie diese überhaupt benötigen. Um dies zu veranschaulichen, betrachten Sie die unten angegebene sehr einfache Matrix-Klasse:

```
template <typename T, std::size_t COL, std::size_t ROW>
class Matrix {
public:
    using value_type = T;

    Matrix() : values(COL * ROW) {}

    static size_t cols() { return COL; }
    static size_t rows() { return ROW; }

    const T& operator()(size_t x, size_t y) const { return values[y * COL + x]; }
    T& operator()(size_t x, size_t y) { return values[y * COL + x]; }

private:
    std::vector<T> values;
};

template <typename T, std::size_t COL, std::size_t ROW>
Matrix<T, COL, ROW>
operator+(const Matrix<T, COL, ROW>& lhs, const Matrix<T, COL, ROW>& rhs)
{
    Matrix<T, COL, ROW> result;

    for (size_t y = 0; y != lhs.rows(); ++y) {
        for (size_t x = 0; x != lhs.cols(); ++x) {
            result(x, y) = lhs(x, y) + rhs(x, y);
        }
    }
    return result;
}
```

In Anbetracht der vorherigen Klassendefinition können Sie jetzt Matrix-Ausdrücke schreiben, beispielsweise:

```
const std::size_t cols = 2000;
const std::size_t rows = 1000;

Matrix<double, cols, rows> a, b, c;

// initialize a, b & c
for (std::size_t y = 0; y != rows; ++y) {
    for (std::size_t x = 0; x != cols; ++x) {
        a(x, y) = 1.0;
        b(x, y) = 2.0;
        c(x, y) = 3.0;
    }
}

Matrix<double, cols, rows> d = a + b + c;  // d(x, y) = 6
```

Wenn Sie den `operator+()` überladen können, erhalten Sie eine Notation, die die natürliche mathematische Notation für Matrizen nachahmt.

Leider ist die vorherige Implementierung im Vergleich zu einer entsprechenden "handgefertigten" Version auch sehr ineffizient.

Um zu verstehen, warum, müssen Sie berücksichtigen, was passiert, wenn Sie einen Ausdruck wie `Matrix d = a + b + c` schreiben. Dies erweitert sich tatsächlich auf `((a + b) + c)` oder `operator+(operator+(a, b), c)` . Mit anderen Worten, die Schleife innerhalb von `operator+()` wird zweimal ausgeführt, während sie in einem einzigen Durchgang problemlos ausgeführt werden konnte. Dies führt auch dazu, dass zwei temporäre Systeme erstellt werden, was die Leistung weiter verschlechtert. Im Wesentlichen haben Sie durch Hinzufügen der Flexibilität, eine Notation in der Nähe des mathematischen Gegenstücks zu verwenden, die `Matrix` Klasse auch sehr ineffizient gemacht.

Beispielsweise können Sie ohne Überladen eines Operators eine wesentlich effizientere Matrixsummierung mit einem einzigen Durchlauf implementieren:

```
template<typename T, std::size_t COL, std::size_t ROW>
Matrix<T, COL, ROW> add3(const Matrix<T, COL, ROW>& a,
                         const Matrix<T, COL, ROW>& b,
                         const Matrix<T, COL, ROW>& c)
{
    Matrix<T, COL, ROW> result;
    for (size_t y = 0; y != ROW; ++y) {
        for (size_t x = 0; x != COL; ++x) {
            result(x, y) = a(x, y) + b(x, y) + c(x, y);
        }
    }
    return result;
}
```

Das vorige Beispiel hat jedoch seine eigenen Nachteile, da es eine viel mehr verschachtelte Schnittstelle für die Matrix-Klasse erstellt (Sie müssten Methoden wie `Matrix::add2()` , `Matrix::AddMultiply()` usw.) in Betracht ziehen.

Lassen Sie uns stattdessen einen Schritt zurückgehen und sehen, wie wir die Überlastung der Bediener anpassen können, um eine effizientere Leistung zu erzielen

Das Problem rührt von der Tatsache her, dass der Ausdruck `Matrix d = a + b + c` zu "eifrig" ausgewertet wird, bevor Sie die Möglichkeit hatten, den gesamten Ausdrucksbaum zu erstellen. Mit anderen Worten, Sie möchten `a + b + c` in einem Durchgang auswerten, und nur dann müssen Sie den resultierenden Ausdruck `d` zuweisen.

Dies ist die Kernidee von Ausdrucksvorlagen: Statt `operator+()` sofort das Ergebnis des Hinzufügens von zwei Matrixinstanzen auszuwerten, gibt es eine *"Ausdrucksvorlage"* für die zukünftige Auswertung zurück, sobald der gesamte Ausdrucksbaum erstellt wurde.

Hier ist zum Beispiel eine mögliche Implementierung für eine Ausdrucksvorlage, die der Summation von 2 Typen entspricht:

```
template <typename LHS, typename RHS>
class MatrixSum
{
```

```
public:
    using value_type = typename LHS::value_type;

    MatrixSum(const LHS& lhs, const RHS& rhs) : rhs(rhs), lhs(lhs) {}

    value_type operator() (int x, int y) const  {
        return lhs(x, y) + rhs(x, y);
    }
private:
    const LHS& lhs;
    const RHS& rhs;
};
```

Und hier ist die aktualisierte Version von `operator+()`

```
template <typename LHS, typename RHS>
MatrixSum<LHS, RHS> operator+(const LHS& lhs, const LHS& rhs) {
    return MatrixSum<LHS, RHS>(lhs, rhs);
}
```

Wie Sie sehen, gibt `operator+()` keine "eifrige Bewertung" des Ergebnisses des Hinzufügens von 2 Matrix-Instanzen (was eine andere Matrix-Instanz wäre) zurück, sondern eine Ausdrucksvorlage, die die Additionsoperation darstellt. Der wichtigste Punkt, den Sie beachten sollten, ist, dass der Ausdruck noch nicht bewertet wurde. Es enthält lediglich Verweise auf seine Operanden.

Tatsächlich `MatrixSum<>` Sie nichts daran, die `MatrixSum<>` Ausdrucksvorlage wie folgt zu instanziieren:

```
MatrixSum<Matrix<double>, Matrix<double> > SumAB(a, b);
```

Sie können den Ausdruck `d = a + b` jedoch zu einem späteren Zeitpunkt, wenn Sie das Ergebnis der Summation tatsächlich benötigen, folgendermaßen bewerten:

```
for (std::size_t y = 0; y != a.rows(); ++y) {
    for (std::size_t x = 0; x != a.cols(); ++x) {
        d(x, y) = SumAB(x, y);
    }
}
```

Wie Sie sehen, besteht ein weiterer Vorteil der Verwendung einer Ausdrucksvorlage darin, dass Sie im Wesentlichen die Summe von `a` und `b` auswerten und sie in einem einzigen Durchgang `d` zuordnen können.

Außerdem hindert Sie nichts daran, mehrere Ausdrucksvorlagen zu kombinieren. Beispiel: `a + b + c` führt zur folgenden Ausdrucksvorlage:

```
MatrixSum<MatrixSum<Matrix<double>, Matrix<double> >, Matrix<double> > SumABC(SumAB, c);
```

Und auch hier können Sie das Endergebnis mit einem einzigen Durchlauf auswerten:

```
for (std::size_t y = 0; y != a.rows(); ++y) {
```

```
    for (std::size_t x = 0; x != a.cols(); ++x) {
        d(x, y) = SumABC(x, y);
    }
}
```

Der letzte Teil des Puzzles besteht schließlich darin, Ihre Ausdrucksvorlage tatsächlich in die `Matrix` Klasse `Matrix` . Dies wird im Wesentlichen durch die Bereitstellung einer Implementierung für `Matrix::operator=()` , die die Ausdrucksvorlage als Argument nimmt und sie in einem Durchgang auswertet, wie Sie es zuvor "manuell" getan haben:

```
template <typename T, std::size_t COL, std::size_t ROW>
class Matrix {
public:
    using value_type = T;

    Matrix() : values(COL * ROW) {}

    static size_t cols() { return COL; }
    static size_t rows() { return ROW; }

    const T& operator()(size_t x, size_t y) const { return values[y * COL + x]; }
    T& operator()(size_t x, size_t y) { return values[y * COL + x]; }

    template <typename E>
    Matrix<T, COL, ROW>& operator=(const E& expression) {
        for (std::size_t y = 0; y != rows(); ++y) {
            for (std::size_t x = 0; x != cols(); ++x) {
                values[y * COL + x] = expression(x, y);
            }
        }
        return *this;
    }

private:
    std::vector<T> values;
};
```

Ausdrucksvorlagen online lesen: https://riptutorial.com/de/cplusplus/topic/3404/ausdrucksvorlagen

Kapitel 12: Ausnahmen

Examples

Ausnahmen fangen

Mit einem `try/catch` Block werden Ausnahmen abgefangen. Der Code im Abschnitt `try` ist der Code, der eine Ausnahme auslösen kann, und der Code in der `catch` Klausel behandelt die Ausnahme.

```
#include <iostream>
#include <string>
#include <stdexcept>

int main() {
  std::string str("foo");

  try {
      str.at(10); // access element, may throw std::out_of_range
  } catch (const std::out_of_range& e) {
      // what() is inherited from std::exception and contains an explanatory message
      std::cout << e.what();
  }
}
```

Mehrere `catch` Klauseln können verwendet werden, um mehrere Ausnahmetypen zu behandeln. Wenn mehrere `catch` Klauseln vorhanden sind, versucht der Ausnahmebehandlungsmechanismus, sie **in der Reihenfolge** ihres Aussehens im Code zu finden:

```
std::string str("foo");

try {
    str.reserve(2); // reserve extra capacity, may throw std::length_error
    str.at(10); // access element, may throw std::out_of_range
} catch (const std::length_error& e) {
    std::cout << e.what();
} catch (const std::out_of_range& e) {
    std::cout << e.what();
}
```

Ausnahmeklassen, die von einer gemeinsamen Basisklasse abgeleitet sind, können mit einer einzigen `catch` Klausel für die allgemeine Basisklasse abgefangen werden. Das obige Beispiel kann die beiden `catch` Klauseln für `std::length_error` und `std::out_of_range` durch eine einzige Klausel für `std:exception` ersetzen:

```
std::string str("foo");

try {
    str.reserve(2); // reserve extra capacity, may throw std::length_error
    str.at(10); // access element, may throw std::out_of_range
```

```
} catch (const std::exception& e) {
    std::cout << e.what();
}
```

Da die `catch` Klauseln in der richtigen Reihenfolge ausprobiert werden, müssen Sie zuerst spezifischere catch-Klauseln schreiben.

```
try {
    /* Code throwing exceptions omitted. */
} catch (const std::exception& e) {
    /* Handle all exceptions of type std::exception. */
} catch (const std::runtime_error& e) {
    /* This block of code will never execute, because std::runtime_error inherits
       from std::exception, and all exceptions of type std::exception were already
       caught by the previous catch clause. */
}
```

Eine andere Möglichkeit ist der Catch-All-Handler, der jedes geworfene Objekt fängt:

```
try {
    throw 10;
} catch (...) {
    std::cout << "caught an exception";
}
```

Ausnahme erneut auslösen

Manchmal möchten Sie etwas mit der Ausnahme tun, die Sie abfangen (z. B. Schreiben, um eine Warnung zu protokollieren oder zu drucken), und es in den oberen Bereich blasen lassen, um behandelt zu werden. Um dies zu tun, können Sie jede Ausnahme erneut auslösen:

```
try {
    ... // some code here
} catch (const SomeException& e) {
    std::cout << "caught an exception";
    throw;
}
```

`throw;` Ohne Argumente wird die aktuell abgefangene Ausnahme erneut ausgegeben.

C ++ 11

Zum Zurückwerfen eines verwalteten `std::exception_ptr` verfügt die C ++ - Standardbibliothek über die Funktion `rethrow_exception` , die verwendet werden kann, indem der Header `<exception>` in Ihr Programm aufgenommen wird.

```
#include <iostream>
#include <string>
#include <exception>
#include <stdexcept>

void handle_eptr(std::exception_ptr eptr) // passing by value is ok
{
```

```
    try {
        if (eptr) {
            std::rethrow_exception(eptr);
        }
    } catch(const std::exception& e) {
        std::cout << "Caught exception \"" << e.what() << "\"\n";
    }
}

int main()
{
    std::exception_ptr eptr;
    try {
        std::string().at(1); // this generates an std::out_of_range
    } catch(...) {
        eptr = std::current_exception(); // capture
    }
    handle_eptr(eptr);
} // destructor for std::out_of_range called here, when the eptr is destructed
```

Funktion Try Blocks In Konstruktor

Die einzige Möglichkeit, eine Ausnahme in der Initialisierungsliste abzufangen:

```
struct A : public B
{
    A() try : B(), foo(1), bar(2)
    {
        // constructor body
    }
    catch (...)
    {
        // exceptions from the initializer list and constructor are caught here
        // if no exception is thrown here
        // then the caught exception is re-thrown.
    }

private:
    Foo foo;
    Bar bar;
};
```

Funktion Try Block für reguläre Funktion

```
void function_with_try_block()
try
{
    // try block body
}
catch (...)
{
    // catch block body
}
```

Welches ist äquivalent zu

```
void function_with_try_block()
{
    try
    {
        // try block body
    }
    catch (...)
    {
        // catch block body
    }
}
```

Beachten Sie, dass das Verhalten von Konstruktoren und Destruktoren anders ist, da der Catch-Block trotzdem eine Ausnahme auslöst (der Catch-Block, wenn es keinen anderen Wurf im Catch-Block-Body gibt).

Die Funktion `main` darf einen Funktionsversuchblock haben wie jede andere Funktion, aber der Funktionsversuchblock des `main` fängt keine Ausnahmen auf, die während der Konstruktion einer nicht lokalen statischen Variablen oder der Zerstörung einer statischen Variablen auftreten. Stattdessen wird `std::terminate` aufgerufen.

Funktion Try Blocks In Destruktor

```
struct A
{
    ~A() noexcept(false) try
    {
        // destructor body
    }
    catch (...)
    {
        // exceptions of destructor body are caught here
        // if no exception is thrown here
        // then the caught exception is re-thrown.
    }
};
```

Beachten Sie, dass dies zwar möglich ist, Sie jedoch beim Abwerfen vom Destruktor sehr vorsichtig sein müssen, als würde ein Destruktor, der beim Stack-Abwickeln aufgerufen wurde, eine Ausnahme `std::terminate` , `std::terminate` aufgerufen.

Bewährte Methode: Nach Wert werfen, nach Konstante fangen

Im Allgemeinen wird es als gute Praxis angesehen, nach Wert (anstatt nach Zeiger) zu werfen, aber durch (const) Referenz zu fangen.

```
try {
    // throw new std::runtime_error("Error!");   // Don't do this!
    // This creates an exception object
    // on the heap and would require you to catch the
    // pointer and manage the memory yourself. This can
    // cause memory leaks!

    throw std::runtime_error("Error!");
```

```
} catch (const std::runtime_error& e) {
    std::cout << e.what() << std::endl;
}
```

Das Fangen durch Referenz ist eine gute Methode, weil es die Notwendigkeit beseitigt, das Objekt zu rekonstruieren, wenn es an den catch-Block übergeben wird (oder bei der Weitergabe an andere catch-Blöcke). Durch das Abfangen nach Verweisen können die Ausnahmen auch polymorph behandelt werden und das Schneiden von Objekten wird vermieden. Wenn Sie jedoch eine Ausnahme erneut `throw e;` (z. B. `throw e;` siehe Beispiel unten), können Sie dennoch Objektschnitte erhalten, da der `throw e;` Die Anweisung erstellt eine Kopie der Ausnahme, wenn der Typ deklariert ist:

```
#include <iostream>

struct BaseException {
    virtual const char* what() const { return "BaseException"; }
};

struct DerivedException : BaseException {
    // "virtual" keyword is optional here
    virtual const char* what() const { return "DerivedException"; }
};

int main(int argc, char** argv) {
    try {
        try {
            throw DerivedException();
        } catch (const BaseException& e) {
            std::cout << "First catch block: " << e.what() << std::endl;
            // Output ==> First catch block: DerivedException

            throw e; // This changes the exception to BaseException
                     // instead of the original DerivedException!
        }
    } catch (const BaseException& e) {
        std::cout << "Second catch block: " << e.what() << std::endl;
        // Output ==> Second catch block: BaseException
    }
    return 0;
}
```

Wenn Sie sicher sind, dass Sie die Ausnahme nicht ändern möchten (z. B. Informationen hinzufügen oder die Nachricht ändern), kann der Compiler durch das Einfangen nach const-Referenz Optimierungen vornehmen und die Leistung verbessern. Dies kann jedoch immer noch zum Splicing von Objekten führen (wie im obigen Beispiel gezeigt).

Warnung: Vermeiden Sie unbeabsichtigte Ausnahmen in `catch` Blöcken, insbesondere im Zusammenhang mit der Zuweisung von zusätzlichem Speicher oder zusätzlichen Ressourcen. Wenn Sie beispielsweise `logic_error` , `runtime_error` oder ihre Unterklassen `logic_error` , kann `runtime_error` `logic_error` , `runtime_error` beim Kopieren der Ausnahmefolge der Speicher ausgeht. E / A-Streams werden möglicherweise während der Protokollierung mit entsprechenden Ausnahmemasken usw. `bad_alloc` .

Verschachtelte Ausnahme

C ++ 11

Während der Ausnahmebehandlung gibt es einen häufigen Anwendungsfall, wenn Sie eine generische Ausnahme von einer Low-Level-Funktion (z. B. einen Dateisystemfehler oder einen Fehler bei der Datenübertragung) abfangen und eine spezifischere High-Level-Ausnahme auslösen, die darauf hinweist, dass ein Vorgang auf höherer Ebene möglich ist nicht ausgeführt werden (z. B., dass ein Foto nicht im Web veröffentlicht werden kann). Dies ermöglicht es der Ausnahmebehandlung, auf spezifische Probleme mit Operationen auf hoher Ebene zu reagieren, und ermöglicht dem Programmierer, nur mit einer Fehlermeldung, einen Ort in der Anwendung zu finden, an dem eine Ausnahme aufgetreten ist. Ein Nachteil dieser Lösung ist, dass der Callstack für Ausnahmen abgeschnitten wird und die ursprüngliche Ausnahmebedingung verloren geht. Dadurch müssen Entwickler den Text der ursprünglichen Ausnahme manuell in eine neu erstellte Ausnahme einfügen.

Verschachtelte Ausnahmen zielen darauf ab, das Problem zu lösen, indem eine Low-Level-Ausnahme, die die Ursache beschreibt, mit einer High-Level-Ausnahme verknüpft wird, die beschreibt, was es in diesem speziellen Fall bedeutet.

`std::nested_exception` können Sie dank `std::throw_with_nested` Ausnahmen `std::throw_with_nested` :

```
#include <stdexcept>
#include <exception>
#include <string>
#include <fstream>
#include <iostream>

struct MyException
{
    MyException(const std::string& message) : message(message) {}
    std::string message;
};

void print_current_exception(int level)
{
    try {
        throw;
    } catch (const std::exception& e) {
        std::cerr << std::string(level, ' ') << "exception: " << e.what() << '\n';
    } catch (const MyException& e) {
        std::cerr << std::string(level, ' ') << "MyException: " << e.message << '\n';
    } catch (...) {
        std::cerr << "Unkown exception\n";
    }
}

void print_current_exception_with_nested(int level =  0)
{
    try {
        throw;
    } catch (...) {
        print_current_exception(level);
    }
    try {
        throw;
    } catch (const std::nested_exception& nested) {
```

```
        try {
            nested.rethrow_nested();
        } catch (...) {
            print_current_exception_with_nested(level + 1); // recursion
        }
    } catch (...) {
        //Empty // End recursion
    }
}

// sample function that catches an exception and wraps it in a nested exception
void open_file(const std::string& s)
{
    try {
        std::ifstream file(s);
        file.exceptions(std::ios_base::failbit);
    } catch(...) {
        std::throw_with_nested(MyException{"Couldn't open " + s});
    }
}

// sample function that catches an exception and wraps it in a nested exception
void run()
{
    try {
        open_file("nonexistent.file");
    } catch(...) {
        std::throw_with_nested( std::runtime_error("run() failed") );
    }
}

// runs the sample function above and prints the caught exception
int main()
{
    try {
        run();
    } catch(...) {
        print_current_exception_with_nested();
    }
}
```

Mögliche Ausgabe:

```
exception: run() failed
 MyException: Couldn't open nonexistent.file
  exception: basic_ios::clear
```

Wenn Sie nur mit Ausnahmen arbeiten, die von `std::exception` exception geerbt wurden, kann der Code sogar vereinfacht werden.

std :: uncaught_exceptions

c ++ 17

C ++ 17 führt `int std::uncaught_exceptions()` (ersetzt das begrenzte `bool std::uncaught_exception()`), um zu erfahren, wie viele Ausnahmen derzeit nicht erfasst werden. Dadurch kann eine Klasse feststellen, ob sie beim Abwickeln eines Stapels zerstört wird oder

nicht.

```
#include <exception>
#include <string>
#include <iostream>

// Apply change on destruction:
// Rollback in case of exception (failure)
// Else Commit (success)
class Transaction
{
public:
    Transaction(const std::string& s) : message(s) {}
    Transaction(const Transaction&) = delete;
    Transaction& operator =(const Transaction&) = delete;
    void Commit() { std::cout << message << ": Commit\n"; }
    void RollBack() noexcept(true) { std::cout << message << ": Rollback\n"; }

    // ...

    ~Transaction() {
        if (uncaughtExceptionCount == std::uncaught_exceptions()) {
            Commit(); // May throw.
        } else { // current stack unwinding
            RollBack();
        }
    }

private:
    std::string message;
    int uncaughtExceptionCount = std::uncaught_exceptions();
};

class Foo
{
public:
    ~Foo() {
        try {
            Transaction transaction("In ~Foo"); // Commit,
                                                // even if there is an uncaught exception
            //...
        } catch (const std::exception& e) {
            std::cerr << "exception/~Foo:" << e.what() << std::endl;
        }
    }
};

int main()
{
    try {
        Transaction transaction("In main"); // RollBack
        Foo foo; // ~Foo commit its transaction.
        //...
        throw std::runtime_error("Error");
    } catch (const std::exception& e) {
        std::cerr << "exception/main:" << e.what() << std::endl;
    }
}
```

Ausgabe:

```
In ~Foo: Commit
In main: Rollback
exception/main:Error
```

Benutzerdefinierte Ausnahme

Sie sollten keine Rohwerte als Ausnahmen werfen, sondern eine der Standardausnahmeklassen verwenden oder eigene erstellen.

Wenn Sie Ihre eigene Ausnahmeklasse von `std::exception` geerbt haben, ist dies eine gute Möglichkeit. Hier ist eine benutzerdefinierte Ausnahmeklasse, die direkt von `std::exception` erbt:

```
#include <exception>

class Except: virtual public std::exception {

protected:

    int error_number;               ///< Error number
    int error_offset;               ///< Error offset
    std::string error_message;      ///< Error message

public:

    /** Constructor (C++ STL string, int, int).
     *  @param msg The error message
     *  @param err_num Error number
     *  @param err_off Error offset
     */
    explicit
    Except(const std::string& msg, int err_num, int err_off):
        error_number(err_num),
        error_offset(err_off),
        error_message(msg)
        {}

    /** Destructor.
     *  Virtual to allow for subclassing.
     */
    virtual ~Except() throw () {}

    /** Returns a pointer to the (constant) error description.
     *  @return A pointer to a const char*. The underlying memory
     *  is in possession of the Except object. Callers must
     *  not attempt to free the memory.
     */
    virtual const char* what() const throw () {
       return error_message.c_str();
    }

    /** Returns error number.
     *  @return #error_number
     */
    virtual int getErrorNumber() const throw() {
        return error_number;
    }

    /**Returns error offset.
```

```
     * @return #error_offset
     */
    virtual int getErrorOffset() const throw() {
        return error_offset;
    }

};
```

Ein Beispiel für einen Wurffang:

```
try {
    throw(Except("Couldn't do what you were expecting", -12, -34));
} catch (const Except& e) {
    std::cout<<e.what()
             <<"\nError number: "<<e.getErrorNumber()
             <<"\nError offset: "<<e.getErrorOffset();
}
```

Da Sie nicht nur nur eine dumme Fehlermeldung ausgeben, sondern auch einige andere Werte, die den Fehler genau darstellen, wird Ihre Fehlerbehandlung wesentlich effizienter und aussagekräftiger.

Es gibt eine Ausnahmeklasse, mit der Sie Fehlermeldungen gut behandeln können:
`std::runtime_error`

Sie können auch von dieser Klasse erben:

```
#include <stdexcept>

class Except: virtual public std::runtime_error {

protected:

    int error_number;               ///< Error number
    int error_offset;               ///< Error offset

public:

    /** Constructor (C++ STL string, int, int).
     *  @param msg The error message
     *  @param err_num Error number
     *  @param err_off Error offset
     */
    explicit
    Except(const std::string& msg, int err_num, int err_off):
        std::runtime_error(msg)
        {
            error_number = err_num;
            error_offset = err_off;

        }

    /** Destructor.
     *  Virtual to allow for subclassing.
     */
    virtual ~Except() throw () {}
```

```
    /** Returns error number.
     *  @return #error_number
     */
    virtual int getErrorNumber() const throw() {
        return error_number;
    }

    /**Returns error offset.
     * @return #error_offset
     */
    virtual int getErrorOffset() const throw() {
        return error_offset;
    }

};
```

Beachten Sie, dass ich die `what()` Funktion nicht von der Basisklasse (`std::runtime_error`) `std::runtime_error` dh wir verwenden die Version der Basisklasse von `what()` . Sie können es überschreiben, wenn Sie weitere Agenda haben.

Ausnahmen online lesen: https://riptutorial.com/de/cplusplus/topic/1354/ausnahmen

Kapitel 13: Ausrichtung

Einführung

Alle Typen in C ++ haben eine Ausrichtung. Dies ist eine Einschränkung für die Speicheradresse, in der Objekte dieses Typs erstellt werden können. Eine Speicheradresse gilt für die Erstellung eines Objekts, wenn das Teilen dieser Adresse durch die Ausrichtung des Objekts eine ganze Zahl ist.

Typenanpassungen sind immer eine Zweierpotenz (einschließlich 1).

Bemerkungen

Der Standard garantiert Folgendes:

- Die Ausrichtungsanforderung eines Typs ist ein Teiler seiner Größe. Eine Klasse mit einer Größe von 16 Byte könnte beispielsweise eine Ausrichtung von 1, 2, 4, 8 oder 16 haben, aber nicht 32. (Wenn die Mitglieder einer Klasse nur 14 Byte groß sind, muss die Klasse jedoch eine Ausrichtungsanforderung haben von 8 fügt der Compiler 2 Padding-Bytes ein, um die Klassengröße auf 16 zu setzen.)

- Die vorzeichenbehafteten und vorzeichenlosen Versionen eines Ganzzahltyps haben dieselbe Ausrichtungsanforderung.
- Ein Zeiger auf `void` hat dieselbe Ausrichtungsanforderung wie ein Zeiger auf `char` .
- Die cv-qualifizierten und die cv-nicht qualifizierten Versionen eines Typs haben dieselbe Ausrichtungsanforderung.

Beachten Sie, dass die Ausrichtung zwar in C ++ 03 vorhanden ist, jedoch erst ab C ++ 11 die Möglichkeit bestand, die Ausrichtung (mithilfe von `alignof`) und die Ausrichtung der Ausrichtung (mithilfe von `alignas`) `alignas` .

Examples

Die Ausrichtung eines Typs abfragen

c ++ 11

Die Ausrichtungsanforderung eines Typs kann mit dem Schlüsselwort `alignof` als unären Operator abgefragt werden. Das Ergebnis ist ein konstanter Ausdruck vom Typ `std::size_t` , *dh* er kann zur Kompilierzeit ausgewertet werden.

```
#include <iostream>
int main() {
    std::cout << "The alignment requirement of int is: " << alignof(int) << '\n';
}
```

Mögliche Ausgabe

> Die Ausrichtungsanforderung von int ist: 4

Bei Anwendung auf ein Array ergibt sich die Ausrichtungsanforderung des Elementtyps. Bei Anwendung auf einen Referenztyp ergibt sich die Ausrichtungsanforderung des referenzierten Typs. (Verweise selbst haben keine Ausrichtung, da sie keine Objekte sind.)

Ausrichtung kontrollieren

C ++ 11

Das Schlüsselwort `alignas` kann verwendet werden, um eine Variable, ein Klassendatenelement, eine Deklaration oder Definition einer Klasse oder eine Deklaration oder Definition einer `alignas` zu erzwingen, falls dies unterstützt wird. Es gibt zwei Formen:

- `alignas(x)` , wobei `x` ein konstanter Ausdruck ist, gibt dem Objekt die Ausrichtung `x` , sofern unterstützt.
- `alignas(T)` , wobei `T` ein Typ ist, gibt der Entität eine Ausrichtung entsprechend der Ausrichtungsanforderung von `T` , d. h., `alignof(T)` , sofern unterstützt.

Wenn mehrere `alignas` auf dieselbe Entität angewendet werden, gilt der strengste.

In diesem Beispiel wird garantiert, dass der `buf` ausgerichtet ist, um ein `int` Objekt aufzunehmen, auch wenn sein Elementtyp `unsigned char buf` ist, die möglicherweise eine schwächere Ausrichtung erfordern.

```
alignas(int) unsigned char buf[sizeof(int)];
new (buf) int(42);
```

`alignas` kann nicht verwendet werden, um einem Typ eine kleinere Ausrichtung zu geben, als der Typ ohne diese Deklaration haben würde:

```
alignas(1) int i; //Il-formed, unless `int` on this platform is aligned to 1 byte.
alignas(char) int j; //Il-formed, unless `int` has the same or smaller alignment than `char`.
```

`alignas` , wenn ein ganzzahliger konstanter Ausdruck angegeben wird, eine gültige Ausrichtung erhalten. Gültige Ausrichtungen sind immer Zweierpotenzen und müssen größer als Null sein. Compiler müssen alle gültigen Alignments bis zum Alignment vom Typ `std::max_align_t` . Sie unterstützen *möglicherweise* größere Ausrichtungen als diese, aber die Unterstützung für die Zuweisung von Speicher für solche Objekte ist begrenzt. Die Obergrenze für Alignments hängt von der Implementierung ab.

C ++ 17 bietet eine direkte Unterstützung in `operator new` für die Zuweisung von Speicher für übergeordnete Typen.

Ausrichtung online lesen: https://riptutorial.com/de/cplusplus/topic/9249/ausrichtung

Kapitel 14: Auto

Bemerkungen

Das Schlüsselwort `auto` ist ein Typname, der einen automatisch abgeleiteten Typ darstellt.

Es war bereits ein reserviertes Schlüsselwort in C ++ 98, das von C geerbt wurde. In alten Versionen von C ++ konnte explizit angegeben werden, dass eine Variable eine automatische Speicherdauer hat:

```
int main()
{
  auto int i = 5; // removing auto has no effect
}
```

Diese alte Bedeutung ist jetzt entfernt.

Examples

Grundlegende Auto-Probe

Das Schlüsselwort `auto` bietet die automatische Ableitung des Typs einer Variablen.

Dies ist besonders praktisch, wenn Sie lange Typnamen verwenden:

```
std::map< std::string, std::shared_ptr< Widget > > table;
// C++98
std::map< std::string, std::shared_ptr< Widget > >::iterator i = table.find( "42" );
// C++11/14/17
auto j = table.find( "42" );
```

mit bereichsbasierter für Schleifen :

```
vector<int> v = {0, 1, 2, 3, 4, 5};
for(auto n: v)
    std::cout << n << ' ';
```

mit lambdas :

```
auto f = [](){ std::cout << "lambda\n"; };
f();
```

um die Wiederholung des Typs zu vermeiden:

```
auto w = std::make_shared< Widget >();
```

um überraschende und unnötige Kopien zu vermeiden:

```
auto myMap = std::map<int,float>();
myMap.emplace(1,3.14);

std::pair<int,float> const& firstPair2 = *myMap.begin();  // copy!
auto const& firstPair = *myMap.begin();  // no copy!
```

Der Grund für die Kopie ist, dass der zurückgegebene Typ tatsächlich `std::pair<const int,float>` !

Auto- und Ausdrucksvorlagen

`auto` kann auch Probleme verursachen, wenn Ausdrucksvorlagen ins Spiel kommen:

```
auto mult(int c) {
    return c * std::valarray<int>{1};
}

auto v = mult(3);
std::cout << v[0]; // some value that could be, but almost certainly is not, 3.
```

Der Grund ist, dass `operator*` on `valarray` Ihnen ein Proxy-Objekt gibt, das sich auf das `valarray` bezieht, um eine `valarray` Bewertung `valarray` . Wenn Sie `auto` , erstellen Sie eine baumelnde Referenz. Statt `mult` hatte `std::valarray<int>` , dann würde der Code definitiv 3 drucken.

auto, const und referenzen

Das `auto` Schlüsselwort selbst repräsentiert einen Werttyp, ähnlich wie `int` oder `char` . Sie kann mit dem Schlüsselwort `const` und dem Symbol `&` geändert werden, um einen konstanten Typ oder einen Referenztyp darzustellen. Diese Modifikatoren können kombiniert werden.

In diesem Beispiel ist `s` ein Wertetyp (sein Typ wird als `std::string`), sodass jede Iteration der `for` Schleife einen String aus dem Vektor in `s` *kopiert* .

```
std::vector<std::string> strings = { "stuff", "things", "misc" };
for(auto s : strings) {
    std::cout << s << std::endl;
}
```

Wenn der Hauptteil der Schleife `s` ändert (z. B. durch Aufrufen von `s.append(" and stuff")`), wird nur diese Kopie geändert, nicht das ursprüngliche `s.append(" and stuff")` von `strings` .

Wenn `s` jedoch mit `auto&` deklariert ist, ist es ein Referenztyp (wird als `std::string&`), so dass ihm bei jeder Wiederholung der Schleife eine *Referenz* auf einen String im Vektor zugewiesen wird:

```
for(auto& s : strings) {
    std::cout << s << std::endl;
}
```

Im Hauptteil dieser Schleife wirken sich Änderungen an `s` direkt auf das Element der `strings` , auf die es verweist.

Wenn `s` als `const auto&` deklariert ist, ist es schließlich ein const-Referenztyp. Dies bedeutet, dass

ihm bei jeder Iteration der Schleife ein *const-Referenzwert* für einen String im Vektor zugewiesen wird:

```
for(const auto& s : strings) {
    std::cout << s << std::endl;
}
```

Innerhalb dieser Schleife können `s` nicht geändert werden (dh es können keine Nicht-Konstanten-Methoden aufgerufen werden).

Wenn Sie `auto` mit bereichsbasierten `for` Schleifen verwenden, ist es im Allgemeinen ratsam, `const auto&` wenn der Schleifenrumpf die überlaufene Struktur nicht ändert, da dadurch unnötige Kopien vermieden werden.

Hinterlegter Rückgabetyp

`auto` wird in der Syntax für abschließende Rückgabetypen verwendet:

```
auto main() -> int {}
```

das ist äquivalent zu

```
int main() {}
```

Meistens nützlich in Verbindung mit `decltype` , um Parameter anstelle von `std::declval<T>` :

```
template <typename T1, typename T2>
auto Add(const T1& lhs, const T2& rhs) -> decltype(lhs + rhs) { return lhs + rhs; }
```

Generisches Lambda (C ++ 14)

C ++ 14

C ++ 14 erlaubt die Verwendung von `auto` in Lambda-Argumenten

```
auto print = [](const auto& arg) { std::cout << arg << std::endl; };

print(42);
print("hello world");
```

Das Lambda entspricht meistens

```
struct lambda {
    template <typename T>
    auto operator ()(const T& arg) const {
        std::cout << arg << std::endl;
    }
};
```

und dann

```
lambda print;

print(42);
print("hello world");
```

Auto- und Proxy-Objekte

Manchmal verhält sich `auto` möglicherweise nicht ganz so, wie es von einem Programmierer erwartet wurde. Der Typ leitet den Ausdruck ab, auch wenn der Typabzug nicht richtig ist.

Wenn zum Beispiel Proxy-Objekte im Code verwendet werden:

```
std::vector<bool> flags{true, true, false};
auto flag = flags[0];
flags.push_back(true);
```

Hier `flag` würde nicht `bool` , aber `std::vector<bool>::reference` , da für `bool` Spezialisierung von Template - `vector` des `operator []` gibt ein Proxy - Objekt mit Konvertierungsoperator `operator bool` definiert.

Wenn `flags.push_back(true)` den Container ändert, kann diese Pseudo-Referenz enden und sich auf ein Element beziehen, das nicht mehr vorhanden ist.

Es macht auch die nächste Situation möglich:

```
void foo(bool b);

std::vector<bool> getFlags();

auto flag = getFlags()[5];
foo(flag);
```

Der `vector` wird sofort gelöscht. `flag` ist also eine Pseudo-Referenz auf ein Element, das gelöscht wurde. Der Aufruf von `foo` ruft ein undefiniertes Verhalten auf.

In solchen Fällen können Sie eine Variable mit `auto` deklarieren und initialisieren, indem Sie den Typ ausführen, den Sie ableiten möchten:

```
auto flag = static_cast<bool>(getFlags()[5]);
```

an diesem Punkt ist es jedoch sinnvoller, `auto` durch `bool` ersetzen.

Ein weiterer Fall, in dem Proxy-Objekte Probleme verursachen können, sind Ausdrucksvorlagen . In diesem Fall sind die Vorlagen manchmal nicht so ausgelegt, dass sie aus Effizienzgründen über den aktuellen Vollausdruck hinausgehen. Die Verwendung des Proxy-Objekts beim nächsten bewirkt ein undefiniertes Verhalten.

Auto online lesen: https://riptutorial.com/de/cplusplus/topic/2421/auto

Kapitel 15: Bauen Sie Systeme auf

Einführung

C ++ hat wie C eine lange und abwechslungsreiche Geschichte in Bezug auf Kompilierungsworkflows und Buildprozesse. Heute verfügt C ++ über verschiedene gängige Build-Systeme, die zum Kompilieren von Programmen verwendet werden, manchmal für mehrere Plattformen innerhalb eines Build-Systems. Hier werden einige Build-Systeme überprüft und analysiert.

Bemerkungen

Derzeit gibt es kein universelles oder dominantes Buildsystem für C ++, das sowohl populär als auch plattformübergreifend ist. Es gibt jedoch mehrere wichtige Build-Systeme, die mit wichtigen Plattformen / Projekten verbunden sind, wobei GNU Make mit dem Betriebssystem GNU / Linux und NMAKE mit dem Projektsystem Visual C ++ / Visual Studio am bemerkenswertesten sind.

Darüber hinaus enthalten einige Integrated Development Environments (IDEs) spezielle Build-Systeme, die speziell für die native IDE verwendet werden. Bestimmte Buildsystemgeneratoren können diese systemeigenen IDE-Buildsystem- / Projektformate generieren, beispielsweise CMake für Eclipse und Microsoft Visual Studio 2012.

Examples

Build-Umgebung mit CMake erstellen

CMake generiert Build-Umgebungen für nahezu jeden Compiler oder IDE aus einer einzigen Projektdefinition. Die folgenden Beispiele zeigen, wie Sie eine CMake-Datei zum plattformübergreifenden C ++ - Code "Hello World" hinzufügen.

CMake-Dateien haben immer den Namen "CMakeLists.txt" und sollten bereits im Stammverzeichnis jedes Projekts (und möglicherweise auch in Unterverzeichnissen) vorhanden sein. Eine grundlegende CMakeLists.txt-Datei sieht folgendermaßen aus:

```
cmake_minimum_required(VERSION 2.4)

project(HelloWorld)

add_executable(HelloWorld main.cpp)
```

Sehen Sie live auf Coliru .

Diese Datei teilt CMake den Projektnamen, die zu erwartende Dateiversion und Anweisungen zum Generieren einer ausführbaren Datei mit dem Namen "HelloWorld" mit, die `main.cpp` erfordert.

Generieren Sie über die Befehlszeile eine Build-Umgebung für Ihren installierten Compiler / IDE:

```
> cmake .
```

Erstellen Sie die Anwendung mit:

```
> cmake --build .
```

Dies generiert die Standard-Build-Umgebung für das System, abhängig vom Betriebssystem und den installierten Tools. Halten Sie den Quellcode von Build-Artefakten mithilfe von Out-of-Source-Builds frei:

```
> mkdir build
> cd build
> cmake ..
> cmake --build .
```

CMake kann auch die grundlegenden Befehle der Plattform-Shell vom vorherigen Beispiel abstrahieren:

```
> cmake -E make_directory build
> cmake -E chdir build cmake ..
> cmake --build build
```

CMake enthält Generatoren für eine Reihe allgemeiner Build-Tools und IDEs. So erstellen Sie Makefiles für `nmake` Visual Studio :

```
> cmake -G "NMake Makefiles" ..
> nmake
```

Kompilieren mit GNU make

Einführung

GNU Make (Styled `make`) ist ein Programm zur Automatisierung der Ausführung von Shell-Befehlen. GNU Make ist ein spezifisches Programm, das unter die Make-Familie fällt. Make ist nach wie vor beliebt bei Unix- und POSIX-ähnlichen Betriebssystemen, einschließlich derjenigen, die vom Linux-Kernel, Mac OS X und BSD stammen.

GNU Make ist besonders bemerkenswert, weil es an das GNU-Projekt angeschlossen ist, das an das beliebte GNU / Linux-Betriebssystem angeschlossen ist. GNU Make hat auch kompatible Versionen, die auf verschiedenen Windows- und Mac OS X-Versionen laufen. Es ist auch eine sehr stabile Version mit historischer Bedeutung, die nach wie vor beliebt ist. Aus diesen Gründen wird GNU Make häufig neben C und C ++ unterrichtet.

Grundregeln

Erstellen Sie zum Kompilieren mit make ein Makefile in Ihrem Projektverzeichnis. Ihr Makefile kann so einfach sein wie:

Makefile

```
# Set some variables to use in our command
# First, we set the compiler to be g++
CXX=g++

# Then, we say that we want to compile with g++'s recommended warnings and some extra ones.
CXXFLAGS=-Wall -Wextra -pedantic

# This will be the output file
EXE=app

SRCS=main.cpp

# When you call `make` at the command line, this "target" is called.
# The $(EXE) at the right says that the `all` target depends on the `$(EXE)` target.
# $(EXE) expands to be the content of the EXE variable
# Note: Because this is the first target, it becomes the default target if `make` is called
without target
all: $(EXE)

# This is equivalent to saying
# app: $(SRCS)
# $(SRCS) can be separated, which means that this target would depend on each file.
# Note that this target has a "method body": the part indented by a tab (not four spaces).
# When we build this target, make will execute the command, which is:
# g++ -Wall -Wextra -pedantic -o app main.cpp
# I.E. Compile main.cpp with warnings, and output to the file ./app
$(EXE): $(SRCS)
    @$(CXX) $(CXXFLAGS) -o $@ $(SRCS)

# This target should reverse the `all` target. If you call
# make with an argument, like `make clean`, the corresponding target
# gets called.
clean:
    @rm -f $(EXE)
```

> **HINWEIS: Stellen Sie absolut sicher, dass die Einrückungen mit einer Registerkarte und nicht mit vier Leerzeichen versehen sind. Andernfalls erhalten Sie eine Fehlermeldung mit** `Makefile:10: *** missing separator. Stop.`

Führen Sie folgende Schritte aus, um dies über die Befehlszeile auszuführen:

```
$ cd ~/Path/to/project
$ make
$ ls
app  main.cpp  Makefile

$ ./app
Hello World!

$ make clean
$ ls
main.cpp  Makefile
```

Inkrementelle Builds

Wenn Sie anfangen, mehr Dateien zu haben, wird Make nützlicher. Was ist, wenn Sie **a.cpp,** aber nicht **b.cpp** bearbeitet **haben** ? Das **Rekompilieren von b.cpp** würde länger dauern.

Mit folgender Verzeichnisstruktur:

```
.
+-- src
|   +-- a.cpp
|   +-- a.hpp
|   +-- b.cpp
|   +-- b.hpp
+-- Makefile
```

Das wäre ein gutes Makefile:

Makefile

```
CXX=g++
CXXFLAGS=-Wall -Wextra -pedantic
EXE=app

SRCS_GLOB=src/*.cpp
SRCS=$(wildcard $(SRCS_GLOB))
OBJS=$(SRCS:.cpp=.o)

all: $(EXE)

$(EXE): $(OBJS)
    @$(CXX) -o $@ $(OBJS)

depend: .depend

.depend: $(SRCS)
    @-rm -f ./.depend
    @$(CXX) $(CXXFLAGS) -MM $^>>./.depend

clean:
    -rm -f $(EXE)
    -rm $(OBJS)
    -rm *~
    -rm .depend

include .depend
```

Beobachte wieder die Tabs. Dieses neue Makefile stellt sicher, dass Sie nur geänderte Dateien neu kompilieren, wodurch die Kompilierzeit minimiert wird.

Dokumentation

Weitere Informationen zu make finden Sie in der offiziellen Dokumentation der Free Software

Foundation , der stackoverflow-Dokumentation und der ausführlichen Antwort von dmckee zu stackoverflow .

Bauen mit SCons

Sie können den plattformübergreifenden C ++ - Code "Hello World" mithilfe von Scons - Ein Python- Programmiersprache-Werkzeug erstellen .

Erstellen Sie zuerst eine Datei namens `SConstruct` (beachten Sie, dass SCons standardmäßig nach einer Datei mit genau diesem Namen sucht). Im `hello.cpp` sollte sich die Datei in einem Verzeichnis entlang Ihrer `hello.cpp` . Schreiben Sie die Zeile in die neue Datei

```
Program('hello.cpp')
```

Führen `scons` nun vom Terminal aus `scons` . Sie sollten so etwas sehen

```
$ scons
scons: Reading SConscript files ...
scons: done reading SConscript files.
scons: Building targets ...
g++ -o hello.o -c hello.cpp
g++ -o hello hello.o
scons: done building targets.
```

(Die Details können jedoch je nach Betriebssystem und installiertem Compiler variieren).

Die Klassen `Environment` und `Glob` helfen Ihnen bei der weiteren Konfiguration der Erstellung. ZB die `SConstruct` Datei

```
env=Environment(CPPPATH='/usr/include/boost/',
    CPPDEFINES=[],
    LIBS=[],
    SCONS_CXX_STANDARD="c++11"
    )

env.Program('hello', Glob('src/*.cpp'))
```

erstellt die ausführbare Datei `hello` und verwendet alle `cpp` Dateien in `src` . Sein `CPPPATH` ist `/usr/include/boost` und gibt den C ++ 11-Standard an.

Ninja

Einführung

Das Ninja-Build-System wird von seiner Projektwebsite als "kleines Build-System mit Fokus auf Geschwindigkeit" beschrieben. Ninja ist darauf ausgelegt, dass seine Dateien von Build-System-Dateigeneratoren generiert werden. Im Gegensatz zu Build-System-Managern auf höherer Ebene wie CMake oder Meson werden die Systeme auf untergeordneter Ebene erstellt.

Ninja ist hauptsächlich in C ++ und Python geschrieben und wurde als Alternative zum SCons-Buildsystem für das Chromium-Projekt erstellt.

NMAKE (Microsoft-Programmverwaltungsprogramm)

Einführung

NMAKE ist ein von Microsoft entwickeltes Befehlszeilenprogramm, das hauptsächlich in Verbindung mit Microsoft Visual Studio und / oder den Visual C ++ - Befehlszeilenprogrammen verwendet wird.

NMAKE ist ein Build-System, das unter die Make-Familie von Build-Systemen fällt, verfügt jedoch über bestimmte, von Unix-ähnlichen Make-Programmen abweichende Funktionen, z.

Autotools (GNU)

Einführung

Die Autotools sind eine Gruppe von Programmen, die ein GNU Build System für ein bestimmtes Softwarepaket erstellen. Es ist eine Reihe von Tools, die zusammenarbeiten, um verschiedene Build-Ressourcen zu erstellen, beispielsweise ein Makefile (zur Verwendung mit GNU Make). Autotools können somit als De-facto-Build-Systemgenerator betrachtet werden.

Einige bemerkenswerte Autotools-Programme umfassen:

- Autoconf
- Automake (nicht mit `make` verwechseln)

Im Allgemeinen soll Autotools das Unix-kompatible Skript und das Makefile generieren, damit der folgende Befehl (im einfachen Fall) die meisten Pakete erstellen und installieren kann:

```
./configure && make && make install
```

Daher hat Autotools auch eine Beziehung zu bestimmten Paketmanagern, insbesondere zu solchen, die an Betriebssysteme angeschlossen sind, die den POSIX-Standards entsprechen.

Bauen Sie Systeme auf online lesen: https://riptutorial.com/de/cplusplus/topic/8200/bauen-sie-systeme-auf

Kapitel 16: Beispiele für Client-Server

Examples

Hallo TCP Server

Lassen Sie mich zunächst sagen, Sie sollten zuerst Beejs Guide to Network Programming besuchen und es kurz durchlesen, was die meisten dieser Dinge etwas ausführlicher erklärt. Wir erstellen hier einen einfachen TCP-Server, der zu allen eingehenden Verbindungen "Hello World" sagt und diese dann schließt. Zu beachten ist auch, dass der Server iterativ mit den Clients kommuniziert, was jeweils einen Client bedeutet. Schauen Sie sich die relevanten Manpages an, da diese wertvolle Informationen zu den einzelnen Funktionsaufrufen und Socketstrukturen enthalten können.

Wir werden den Server mit einem Port betreiben, also nehmen wir auch ein Argument für die Portnummer. Lass uns mit Code anfangen -

```
#include <cstring>    // sizeof()
#include <iostream>
#include <string>

// headers for socket(), getaddrinfo() and friends
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>

#include <unistd.h>    // close()


int main(int argc, char *argv[])
{
    // Let's check if port number is supplied or not..
    if (argc != 2) {
        std::cerr << "Run program as 'program <port>'\n";
        return -1;
    }

    auto &portNum = argv[1];
    const unsigned int backLog = 8;  // number of connections allowed on the incoming queue


    addrinfo hints, *res, *p;    // we need 2 pointers, res to hold and p to iterate over
    memset(&hints, 0, sizeof(hints));

    // for more explanation, man socket
    hints.ai_family   = AF_UNSPEC;    // don't specify which IP version to use yet
    hints.ai_socktype = SOCK_STREAM;  // SOCK_STREAM refers to TCP, SOCK_DGRAM will be?
    hints.ai_flags    = AI_PASSIVE;


    // man getaddrinfo
    int gAddRes = getaddrinfo(NULL, portNum, &hints, &res);
    if (gAddRes != 0) {
```

```
        std::cerr << gai_strerror(gAddRes) << "\n";
        return -2;
    }

    std::cout << "Detecting addresses" << std::endl;

    unsigned int numOfAddr = 0;
    char ipStr[INET6_ADDRSTRLEN];    // ipv6 length makes sure both ipv4/6 addresses can be
stored in this variable


    // Now since getaddrinfo() has given us a list of addresses
    // we're going to iterate over them and ask user to choose one
    // address for program to bind to
    for (p = res; p != NULL; p = p->ai_next) {
        void *addr;
        std::string ipVer;

        // if address is ipv4 address
        if (p->ai_family == AF_INET) {
            ipVer             = "IPv4";
            sockaddr_in *ipv4 = reinterpret_cast<sockaddr_in *>(p->ai_addr);
            addr              = &(ipv4->sin_addr);
            ++numOfAddr;
        }

        // if address is ipv6 address
        else {
            ipVer              = "IPv6";
            sockaddr_in6 *ipv6 = reinterpret_cast<sockaddr_in6 *>(p->ai_addr);
            addr               = &(ipv6->sin6_addr);
            ++numOfAddr;
        }

        // convert IPv4 and IPv6 addresses from binary to text form
        inet_ntop(p->ai_family, addr, ipStr, sizeof(ipStr));
        std::cout << "(" << numOfAddr << ") " << ipVer << " : " << ipStr
                  << std::endl;
    }

    // if no addresses found :(
    if (!numOfAddr) {
        std::cerr << "Found no host address to use\n";
        return -3;
    }

    // ask user to choose an address
    std::cout << "Enter the number of host address to bind with: ";
    unsigned int choice = 0;
    bool madeChoice     = false;
    do {
        std::cin >> choice;
        if (choice > (numOfAddr + 1) || choice < 1) {
            madeChoice = false;
            std::cout << "Wrong choice, try again!" << std::endl;
        } else
            madeChoice = true;
    } while (!madeChoice);
```

```
    p = res;

    // let's create a new socket, socketFD is returned as descriptor
    // man socket for more information
    // these calls usually return -1 as result of some error
    int sockFD = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
    if (sockFD == -1) {
        std::cerr << "Error while creating socket\n";
        freeaddrinfo(res);
        return -4;
    }


    // Let's bind address to our socket we've just created
    int bindR = bind(sockFD, p->ai_addr, p->ai_addrlen);
    if (bindR == -1) {
        std::cerr << "Error while binding socket\n";

        // if some error occurs, make sure to close socket and free resources
        close(sockFD);
        freeaddrinfo(res);
        return -5;
    }


    // finally start listening for connections on our socket
    int listenR = listen(sockFD, backLog);
    if (listenR == -1) {
        std::cerr << "Error while Listening on socket\n";

        // if some error occurs, make sure to close socket and free resources
        close(sockFD);
        freeaddrinfo(res);
        return -6;
    }


    // structure large enough to hold client's address
    sockaddr_storage client_addr;
    socklen_t client_addr_size = sizeof(client_addr);


    const std::string response = "Hello World";


    // a fresh infinite loop to communicate with incoming connections
    // this will take client connections one at a time
    // in further examples, we're going to use fork() call for each client connection
    while (1) {

        // accept call will give us a new socket descriptor
        int newFD
          = accept(sockFD, (sockaddr *) &client_addr, &client_addr_size);
        if (newFD == -1) {
            std::cerr << "Error while Accepting on socket\n";
            continue;
        }

        // send call sends the data you specify as second param and it's length as 3rd param,
also returns how many bytes were actually sent
        auto bytes_sent = send(newFD, response.data(), response.length(), 0);
```

```
        close(newFD);
    }

    close(sockFD);
    freeaddrinfo(res);

    return 0;
}
```

Das folgende Programm läuft als -

```
Detecting addresses
(1) IPv4 : 0.0.0.0
(2) IPv6 : ::
Enter the number of host address to bind with: 1
```

Hallo TCP-Client

Dieses Programm ist ein komplettes Hello TCP Server-Programm. Sie können eines der beiden Programme ausführen, um die Gültigkeit des anderen zu überprüfen. Der Programmablauf ist bei Hello TCP-Servern durchaus üblich. Sehen Sie sich auch diesen an.

Hier ist der Code -

```
#include <cstring>
#include <iostream>
#include <string>

#include <arpa/inet.h>
#include <netdb.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <unistd.h>


int main(int argc, char *argv[])
{
    // Now we're taking an ipaddress and a port number as arguments to our program
    if (argc != 3) {
        std::cerr << "Run program as 'program <ipaddress> <port>'\n";
        return -1;
    }


    auto &ipAddress = argv[1];
    auto &portNum   = argv[2];

    addrinfo hints, *p;
    memset(&hints, 0, sizeof(hints));
    hints.ai_family   = AF_UNSPEC;
    hints.ai_socktype = SOCK_STREAM;
    hints.ai_flags    = AI_PASSIVE;

    int gAddRes = getaddrinfo(ipAddress, portNum, &hints, &p);
    if (gAddRes != 0) {
        std::cerr << gai_strerror(gAddRes) << "\n";
        return -2;
```

```
    }

    if (p == NULL) {
        std::cerr << "No addresses found\n";
        return -3;
    }

    // socket() call creates a new socket and returns it's descriptor
    int sockFD = socket(p->ai_family, p->ai_socktype, p->ai_protocol);
    if (sockFD == -1) {
        std::cerr << "Error while creating socket\n";
        return -4;
    }

    // Note: there is no bind() call as there was in Hello TCP Server
    // why? well you could call it though it's not necessary
    // because client doesn't necessarily has to have a fixed port number
    // so next call will bind it to a random available port number

    // connect() call tries to establish a TCP connection to the specified server
    int connectR = connect(sockFD, p->ai_addr, p->ai_addrlen);
    if (connectR == -1) {
        close(sockFD);
        std::cerr << "Error while connecting socket\n";
        return -5;
    }

    std::string reply(15, ' ');

    // recv() call tries to get the response from server
    // BUT there's a catch here, the response might take multiple calls
    // to recv() before it is completely received
    // will be demonstrated in another example to keep this minimal
    auto bytes_recv = recv(sockFD, &reply.front(), reply.size(), 0);
    if (bytes_recv == -1) {
        std::cerr << "Error while receiving bytes\n";
        return -6;
    }

    std::cout << "\nClient recieved: " << reply << std::endl;
    close(sockFD);
    freeaddrinfo(p);

    return 0;
}
```

Beispiele für Client-Server online lesen: https://riptutorial.com/de/cplusplus/topic/7177/beispiele-fur-client-server

Kapitel 17: Benutzerdefinierte Literale

Examples

Benutzerdefinierte Literale mit langen Doppelwerten

```
#include <iostream>

long double operator"" _km(long double val)
{
    return val * 1000.0;
}

long double operator"" _mi(long double val)
{
    return val * 1609.344;
}

int main()
{
    std::cout << "3 km = " << 3.0_km << " m\n";
    std::cout << "3 mi = " << 3.0_mi << " m\n";
    return 0;
}
```

Die Ausgabe dieses Programms ist folgende:

```
3 km = 3000 m
3 mi = 4828.03 m
```

Benutzerdefinierte Standardliterale für die Dauer

C ++ 14

Die folgenden Benutzerliterale der Dauer werden im `namespace std::literals::chrono_literals` , wobei sowohl `literals` als auch `chrono_literals` Inline-Namespaces sind . Der Zugang zu diesen Operatoren können mit gewonnen werden `using namespace std::literals` , `using namespace std::chrono_literals` und `using namespace std::literals::chrono_literals` .

```
#include <chrono>
#include <iostream>

int main()
{
    using namespace std::literals::chrono_literals;

    std::chrono::nanoseconds t1 = 600ns;
    std::chrono::microseconds t2 = 42us;
    std::chrono::milliseconds t3 = 51ms;
    std::chrono::seconds t4 = 61s;
    std::chrono::minutes t5 = 88min;
    auto t6 = 2 * 0.5h;
```

```
    auto total = t1 + t2 + t3 + t4 + t5 + t6;

    std::cout.precision(13);
    std::cout << total.count() << " nanoseconds" << std::endl; // 8941051042600 nanoseconds
    std::cout << std::chrono::duration_cast<std::chrono::hours>(total).count()
              << " hours" << std::endl; // 2 hours
}
```

Benutzerdefinierte Standardliterale für Zeichenfolgen

C ++ 14

Die folgenden String-Benutzerliterale werden im `namespace std::literals::string_literals` , wobei sowohl `literals` als auch `string_literals` Inline-Namespaces sind . Der Zugang zu diesen Operatoren können mit gewonnen werden `using namespace std::literals` , `using namespace std::string_literals` und `using namespace std::literals::string_literals` .

```
#include <codecvt>
#include <iostream>
#include <locale>
#include <string>

int main()
{
    using namespace std::literals::string_literals;

    std::string s = "hello world"s;
    std::u16string s16 = u"hello world"s;
    std::u32string s32 = U"hello world"s;
    std::wstring ws = L"hello world"s;

    std::cout << s << std::endl;

    std::wstring_convert<std::codecvt_utf8_utf16<char16_t>, char16_t> utf16conv;
    std::cout << utf16conv.to_bytes(s16) << std::endl;

    std::wstring_convert<std::codecvt_utf8_utf16<char32_t>, char32_t> utf32conv;
    std::cout << utf32conv.to_bytes(s32) << std::endl;

    std::wcout << ws << std::endl;
}
```

Hinweis:

Die wörtliche Zeichenfolge kann `\0`

```
std::string s1 = "foo\0\0bar";  // constructor from C-string: results in "foo"s
std::string s2 = "foo\0\0bar"s; // That string contains 2 '\0' in its middle
```

Benutzerdefinierte Standardliterale für komplexe

C ++ 14

Die folgenden komplexen Benutzerliterale werden im `namespace std::literals::complex_literals` , wobei sowohl `literals` als auch `complex_literals` Inline-Namespaces sind . Der Zugang zu diesen Operatoren können mit gewonnen werden `using namespace std::literals` , `using namespace std::complex_literals` und `using namespace std::literals::complex_literals` .

```
#include <complex>
#include <iostream>

int main()
{
    using namespace std::literals::complex_literals;

    std::complex<double> c = 2.0 + 1i;        // {2.0, 1.}
    std::complex<float> cf = 2.0f + 1if;      // {2.0f, 1.f}
    std::complex<long double> cl = 2.0L + 1il; // {2.0L, 1.L}

    std::cout << "abs" << c << " = " << abs(c) << std::endl;   // abs(2,1) = 2.23607
    std::cout << "abs" << cf << " = " << abs(cf) << std::endl; // abs(2,1) = 2.23607
    std::cout << "abs" << cl << " = " << abs(cl) << std::endl; // abs(2,1) = 2.23607
}
```

Eigenes benutzerdefiniertes Literal für binär

Trotzdem können Sie eine binäre Zahl in C ++ 14 schreiben:

```
int number =0b0001'0101; // ==21
```

Hier ein bekanntes Beispiel mit einer selbst erstellten Implementierung für Binärzahlen:

Hinweis: Das gesamte Vorlagenerweiterungsprogramm wird zur Kompilierzeit ausgeführt.

```
template< char FIRST, char... REST > struct binary
{
    static_assert( FIRST == '0' || FIRST == '1', "invalid binary digit" ) ;
    enum { value = ( ( FIRST - '0' ) << sizeof...(REST) ) + binary<REST...>::value  } ;
};

template<> struct binary<'0'> { enum { value = 0 } ; };
template<> struct binary<'1'> { enum { value = 1 } ; };


// raw literal operator
template<  char... LITERAL > inline
constexpr unsigned int operator "" _b() { return binary<LITERAL...>::value ; }

// raw literal operator
template<  char... LITERAL > inline
constexpr unsigned int operator "" _B() { return binary<LITERAL...>::value ; }

#include <iostream>

int main()
{
    std::cout  << 10101_B << ", " << 011011000111_b  << '\n' ; // prints 21, 1735
}
```

Benutzerdefinierte Literale online lesen:
https://riptutorial.com/de/cplusplus/topic/2745/benutzerdefinierte-literale

Kapitel 18: Bereiche

Examples

Einfacher Blockumfang

Der Gültigkeitsbereich einer Variablen in einem Block `{ ... }` beginnt nach der Deklaration und endet am Ende des Blocks. Wenn es einen verschachtelten Block gibt, kann der innere Block den Gültigkeitsbereich einer im äußeren Block deklarierten Variablen verbergen.

```
{
    int x = 100;
    //   ^
    //   Scope of `x` begins here
    //
}   // <- Scope of `x` ends here
```

Wenn ein verschachtelter Block innerhalb eines äußeren Blocks beginnt, verbirgt eine neue deklarierte Variable mit demselben Namen, die zuvor in der äußeren Klasse war, die erste.

```
{
    int x = 100;

    {
        int x = 200;

        std::cout << x;  // <- Output is 200
    }

    std::cout << x;  // <- Output is 100
}
```

Globale Variablen

Um eine einzelne Instanz einer Variablen zu deklarieren, auf die in verschiedenen Quelldateien zugegriffen werden kann, ist es möglich, sie mit dem Schlüsselwort `extern` im globalen Bereich zu `extern` . Dieses Schlüsselwort sagt dem Compiler, dass irgendwo im Code eine Definition für diese Variable vorhanden ist, sodass sie überall verwendet werden kann und dass alle Schreib- / Lesevorgänge an einer Stelle des Speichers erfolgen.

```
// File my_globals.h:

#ifndef __MY_GLOBALS_H__
#define __MY_GLOBALS_H__

extern int circle_radius; // Promise to the compiler that circle_radius
                          // will be defined somewhere

#endif
```

```
// File foo1.cpp:

#include "my_globals.h"

int circle_radius = 123; // Defining the extern variable
```

```
// File main.cpp:

#include "my_globals.h"
#include <iostream>

int main()
{
    std::cout << "The radius is: " << circle_radius << "\n";'
    return 0;
}
```

Ausgabe:

```
The radius is: 123
```

Bereiche online lesen: https://riptutorial.com/de/cplusplus/topic/3453/bereiche

Kapitel 19: Bitfelder

Einführung

Bitfelder fassen C- und C ++ - Strukturen eng zusammen, um die Größe zu reduzieren. Dies erscheint mühelos: Geben Sie die Anzahl der Bits für die Member an, und der Compiler erledigt das Zusammenmischen von Bits. Die Einschränkung besteht darin, dass die Adresse eines Bitfeldmitglieds nicht übernommen werden kann, da sie gemischt gespeichert wird. `sizeof()` ist ebenfalls nicht zulässig.

Die Kosten für Bitfelder sind langsamer, da Speicher abgerufen und bitweise Operationen zum Extrahieren oder Ändern von Elementwerten ausgeführt werden müssen. Diese Vorgänge erhöhen auch die Größe der ausführbaren Datei.

Bemerkungen

Wie teuer sind die bitweisen Operationen? Angenommen, eine einfache Nicht-Bit-Feldstruktur:

```
struct foo {
    unsigned x;
    unsigned y;
}
static struct foo my_var;
```

In einem späteren Code:

```
my_var.y = 5;
```

Wenn `sizeof (unsigned) == 4` , wird x am Anfang der Struktur gespeichert, und y wird 4 Byte in gespeichert. Der erzeugte Assembly-Code kann folgendermaßen aussehen:

```
loada register1,#myvar     ; get the address of the structure
storei register1[4],#0x05  ; put the value '5' at offset 4, e.g., set y=5
```

Dies ist einfach, weil x nicht mit y vermischt ist. Stellen Sie sich jedoch vor, Sie definieren die Struktur mit Bitfeldern neu:

```
struct foo {
    unsigned x : 4; /* Range 0-0x0f, or 0 through 15 */
    unsigned y : 4;
}
```

Sowohl `x` als auch `y` werden 4 Bits zugewiesen, die sich ein einzelnes Byte teilen. Die Struktur beansprucht somit 1 Byte statt 8. Betrachten Sie die Assembly jetzt mit `y` , sofern sie im oberen Halbbyte endet:

```
loada  register1,#myvar        ; get the address of the structure
loadb  register2,register1[0]  ; get the byte from memory
andb   register2,#0x0f         ; zero out y in the byte, leaving x alone
orb    register2,#0x50         ; put the 5 into the 'y' portion of the byte
stb    register1[0],register2  ; put the modified byte back into memory
```

Dies kann ein guter Kompromiss sein, wenn es Tausende oder Millionen solcher Strukturen gibt, und es hilft, den Arbeitsspeicher im Cache zu halten oder ein Auswechseln zu verhindern - oder die ausführbare Datei könnte aufgebläht werden, um diese Probleme zu verschlimmern und die Verarbeitung zu verlangsamen. Wie bei allen Dingen verwenden Sie ein gutes Urteilsvermögen.

Verwendung von Gerätetreibern: Vermeiden Sie Bitfelder als clevere Implementierungsstrategie für Gerätetreiber. Bitfeldspeicher-Layouts sind zwischen Compilern nicht notwendigerweise konsistent, sodass solche Implementierungen nicht portierbar sind. Die Werte zum Lesen, Ändern und Schreiben zum Setzen von Werten werden möglicherweise nicht den Anforderungen der Geräte entsprechen und verursachen unerwartetes Verhalten.

Examples

Erklärung und Verwendung

```
struct FileAttributes
{
    unsigned int ReadOnly: 1;
    unsigned int Hidden: 1;
};
```

Hier belegt jedes dieser beiden Felder 1 Bit im Speicher. Es wird angegeben durch `: 1` Ausdruck hinter den Variablennamen. Der Basistyp des Bitfelds kann ein beliebiger integraler Typ sein (8-Bit-Int. Bis 64-Bit-Int). Die Verwendung eines `unsigned` Typs wird empfohlen. Andernfalls können Überraschungen auftreten.

Wenn mehr Bits erforderlich sind, ersetzen Sie "1" durch die Anzahl der erforderlichen Bits. Zum Beispiel:

```
struct Date
{
    unsigned int Year : 13; // 2^13 = 8192, enough for "year" representation for long time
    unsigned int Month: 4;  // 2^4 = 16, enough to represent 1-12 month values.
    unsigned int Day:   5;  // 32
};
```

Die gesamte Struktur verwendet nur 22 Bit. Bei normalen Compilereinstellungen wäre die `sizeof` dieser Struktur 4 Byte.

Die Nutzung ist ziemlich einfach. Deklarieren Sie einfach die Variable und verwenden Sie sie wie eine gewöhnliche Struktur.

```
Date d;
```

```
d.Year = 2016;
d.Month = 7;
d.Day =  22;

std::cout << "Year:" << d.Year << std::endl <<
        "Month:" << d.Month << std::endl <<
        "Day:" << d.Day << std::endl;
```

Bitfelder online lesen: https://riptutorial.com/de/cplusplus/topic/2710/bitfelder

Kapitel 20: Bit-Manipulation

Bemerkungen

Um `std::bitset` Sie den Header `<bitset>` einfügen .

```
#include <bitset>
```

`std::bitset` überladen alle Operatorfunktionen, um dieselbe Verwendung wie die Behandlung von Bitsätzen im c-Stil zu ermöglichen.

Verweise

- Ein bisschen zwielichtige Hacks

Examples

Ein bisschen einstellen

Bit-Manipulation im C-Stil

Ein Bit kann mit dem bitweisen OR-Operator (`|`) gesetzt werden.

```
// Bit x will be set
number |= 1LL << x;
```

Verwenden von std :: bitset

`set(x)` oder `set(x,true)` - setzt das Bit an Position `x` auf `1` .

```
std::bitset<5> num(std::string("01100"));
num.set(0);      // num is now 01101
num.set(2);      // num is still 01101
num.set(4,true); // num is now 11110
```

Ein bisschen klären

Bit-Manipulation im C-Stil

Ein Bit kann mit dem bitweisen AND-Operator (`&`) gelöscht werden.

```
// Bit x will be cleared
number &= ~(1LL << x);
```

Verwenden von std :: bitset

`reset(x)` oder `set(x,false)` - löscht das Bit an Position `x` .

```
std::bitset<5> num(std::string("01100"));
num.reset(2);     // num is now 01000
num.reset(0);     // num is still 01000
num.set(3,false); // num is now 00000
```

Ein bisschen umschalten

Bit-Manipulation im C-Stil

Ein Bit kann mit dem XOR-Operator (`^`) umgeschaltet werden.

```
// Bit x will be the opposite value of what it is currently
number ^= 1LL << x;
```

Verwenden von std :: bitset

```
std::bitset<4> num(std::string("0100"));
num.flip(2); // num is now 0000
num.flip(0); // num is now 0001
num.flip();  // num is now 1110 (flips all bits)
```

Ein bisschen überprüfen

Bit-Manipulation im C-Stil

Der Wert des Bits kann erhalten werden, indem die Anzahl `x` mal nach rechts verschoben wird und dann bitweise UND (`&`) darauf ausgeführt wird:

```
(number >> x) & 1LL;  // 1 if the 'x'th bit of 'number' is set, 0 otherwise
```

Die Rechtsverschiebungsoperation kann entweder als arithmetische (vorzeichenbehaftete) Verschiebung oder als logische (vorzeichenlose) Verschiebung implementiert werden. Wenn `number` im Ausdruck `number >> x` einen vorzeichenbehafteten Typ und einen negativen Wert hat, ist der resultierende Wert durch die Implementierung definiert.

Wenn wir den Wert dieses Bits direkt an Ort und Stelle benötigen, können Sie stattdessen die

Maske nach links verschieben:

```
(number & (1LL << x));  // (1 << x) if the 'x'th bit of 'number' is set, 0 otherwise
```

Beides kann als Bedingung verwendet werden, da alle Werte ungleich Null als wahr betrachtet werden.

Verwenden von std :: bitset

```
std::bitset<4> num(std::string("0010"));
bool bit_val = num.test(1);  // bit_val value is set to true;
```

Ändern des n-ten Bits in x

Bit-Manipulation im C-Stil

```
// Bit n will be set if x is 1 and cleared if x is 0.
number ^= (-x ^ number) & (1LL << n);
```

Verwenden von std :: bitset

`set(n,val)` - setzt Bit `n` auf den Wert `val` .

```
std::bitset<5> num(std::string("00100"));
num.set(0,true);  // num is now 00101
num.set(2,false); // num is now 00001
```

Setze alle Bits

Bit-Manipulation im C-Stil

```
x = -1; // -1 == 1111 1111 ... 1111b
```

(Hier erfahren Sie, warum dies funktioniert und tatsächlich der beste Ansatz ist.)

Verwenden von std :: bitset

```
std::bitset<10> x;
x.set(); // Sets all bits to '1'
```

Bit-Manipulation im C-Stil

```
template <typename T>
T rightmostSetBitRemoved(T n)
{
    // static_assert(std::is_integral<T>::value && !std::is_signed<T>::value, "type should be
unsigned"); // For c++11 and later
    return n & (n - 1);
}
```

Erläuterung

- Wenn `n` Null ist, haben wir `0 & 0xFF..FF` die Null ist
- ansonsten kann `n` `0bxxxxxx10..00` geschrieben `0bxxxxxx10..00` und `n - 1` ist `0bxxxxxx011..11` , also ist `n & (n - 1)` `0bxxxxxx000..00` .

Gesetzte Bits zählen

Die Populationszählung einer Bitkette wird häufig in der Kryptographie und anderen Anwendungen benötigt, und das Problem wurde umfassend untersucht.

Der naive Weg erfordert eine Iteration pro Bit:

```
unsigned value = 1234;
unsigned bits = 0;  // accumulates the total number of bits set in `n`

for (bits = 0; value; value >>= 1)
  bits += value & 1;
```

Ein schöner Trick (basierend auf Remove-rechtem gesetztem Bit) ist:

```
unsigned bits = 0;  // accumulates the total number of bits set in `n`

for (; value; ++bits)
  value &= value - 1;
```

Es durchläuft so viele Iterationen, wie gesetzte Bits vorhanden sind. Es ist also gut, wenn erwartet wird, dass der `value` wenige Nicht-Null-Bits hat.

Die Methode wurde zuerst von Peter Wegner (in CACM 3/322 - 1960) vorgeschlagen und ist seit ihrer Veröffentlichung in der *Programmiersprache C* von Brian W. Kernighan und Dennis M. Ritchie bekannt.

Dies erfordert 12 arithmetische Operationen, von denen eine eine Multikation ist:

```
unsigned popcount(std::uint64_t x)
{
  const std::uint64_t m1  = 0x5555555555555555;  // binary: 0101...
  const std::uint64_t m2  = 0x3333333333333333;  // binary: 00110011..
  const std::uint64_t m4  = 0x0f0f0f0f0f0f0f0f;  // binary: 0000111100001111

  x -= (x >> 1) & m1;             // put count of each 2 bits into those 2 bits
  x = (x & m2) + ((x >> 2) & m2); // put count of each 4 bits into those 4 bits
  x = (x + (x >> 4)) & m4;        // put count of each 8 bits into those 8 bits
  return (x * h01) >> 56;  // left 8 bits of x + (x<<8) + (x<<16) + (x<<24) + ...
}
```

Diese Art der Implementierung weist das beste Worst-Case-Verhalten auf (weitere Einzelheiten finden Sie unter Hamming-Gewicht).

Viele CPUs verfügen über eine bestimmte Anweisung (z. B. `popcnt` x86), und der Compiler könnte eine spezifische (**nicht standardmäßige**) eingebaute Funktion bieten. ZB mit g ++ gibt es:

```
int __builtin_popcount (unsigned x);
```

Prüfen Sie, ob eine ganze Zahl eine Potenz von 2 ist

Der Trick `n & (n - 1)` (siehe rechtes gesetztes Bit ganz rechts entfernen) ist auch nützlich, um zu bestimmen, ob eine Ganzzahl eine Potenz von 2 ist:

```
bool power_of_2 = n && !(n & (n - 1));
```

Beachten Sie, dass `0` ohne den ersten Teil der Prüfung (`n &&`) fälschlicherweise als Potenz von 2 betrachtet wird.

Bit-Manipulationsanwendung: Klein- und Großbuchstabe

Eine von mehreren Anwendungen der Bitmanipulation ist das Konvertieren eines Buchstabens von klein nach groß oder umgekehrt, indem eine **Maske** und eine richtige **Bitoperation ausgewählt werden** . Zum Beispiel hat das Schreiben diese binäre Darstellung `01(1)00001` , während das Kapitalgegenstück `01(0)00001` , `01(0)00001` . Sie unterscheiden sich lediglich in den Bits in Klammern. In diesem Fall setzt das Konvertieren **eines** Buchstabens von klein nach groß das Bit in Klammern grundsätzlich auf eins. Dazu machen wir folgendes:

```
/**************************************
convert small letter to captial letter.
**************************************
     a: 01100001
  mask: 11011111  <-- (0xDF)  11(0)11111
      :---------
a&mask: 01000001  <-- A letter
**************************************/
```

Der Code zum Konvertieren eines Briefs in einen Brief lautet

```
#include <cstdio>

int main()
{
    char op1 = 'a';  // "a" letter (i.e. small case)
    int mask = 0xDF; // choosing a proper mask

    printf("a (AND) mask = A\n");
    printf("%c   &   0xDF = %c\n", op1, op1 & mask);

    return 0;
}
```

Das Ergebnis ist

```
$ g++ main.cpp -o test1
$ ./test1
a (AND) mask = A
a   &   0xDF = A
```

Bit-Manipulation online lesen: https://riptutorial.com/de/cplusplus/topic/3016/bit-manipulation

Kapitel 21: Bitoperatoren

Bemerkungen

Bit-Shift-Operationen sind nicht für alle Prozessorarchitekturen portierbar. Unterschiedliche Prozessoren können unterschiedliche Bitbreiten haben. Mit anderen Worten, wenn Sie geschrieben haben

```
int a = ~0;
int b = a << 1;
```

Dieser Wert unterscheidet sich auf einer 64-Bit-Maschine gegenüber einer 32-Bit-Maschine oder von einem x86-basierten Prozessor zu einem PIC-basierten Prozessor.

Die Endianität muss nicht für die bitweisen Operationen selbst berücksichtigt werden, das heißt, die Verschiebung nach rechts (`>>`) verschiebt die Bits in Richtung des niedrigstwertigen Bits, und ein XOR führt eine Exklusivität oder die Bits aus. Endian-Ness muss nur bei den Daten selbst berücksichtigt werden. Wenn also die Endian-Ness für Ihre Anwendung ein Problem ist, ist dies unabhängig von bitweisen Operationen ein Problem.

Examples

& - bitweise AND

```
int a = 6;     // 0110b  (0x06)
int b = 10;    // 1010b  (0x0A)
int c = a & b; // 0010b  (0x02)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Ausgabe

```
a = 6, b = 10, c = 2
```

Warum

Ein bisschen weise `AND` arbeitet auf Bitebene und verwendet die folgende boolesche Wahrheitstabelle:

```
TRUE  AND TRUE  = TRUE
TRUE  AND FALSE = FALSE
FALSE AND FALSE = FALSE
```

Wenn der Binärwert für `a` (`0110`) und der Binärwert für `b` (`1010`) `AND` -verknüpft sind, erhalten wir den Binärwert von `0010` :

```
int a = 0 1 1 0
```

```
int b = 1 0 1 0 &
        ---------
int c = 0 0 1 0
```

Das bitweise UND-Element ändert den Wert der ursprünglichen Werte nicht, es sei denn, er ist dem bitweisen Zuweisungsoperator `&=` ausdrücklich zugewiesen:

```
int a = 5;  // 0101b  (0x05)
a &= 10;    // a = 0101b & 1010b
```

| - bitweise ODER

```
int a = 5;     // 0101b  (0x05)
int b = 12;    // 1100b  (0x0C)
int c = a | b; // 1101b  (0x0D)

std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Ausgabe

```
a = 5, b = 12, c = 13
```

Warum

Ein bisschen weise `OR` arbeitet auf Bitebene und verwendet die folgende boolesche Wahrheitstabelle:

```
true OR true = true
true OR false = true
false OR false = false
```

Wenn der Binärwert für `a` (`0101`) und der Binärwert für `b` (`1100`) zusammen `OR` , erhalten wir den Binärwert von `1101` :

```
int a = 0 1 0 1
int b = 1 1 0 0 |
        ---------
int c = 1 1 0 1
```

Das bitweise ODER ändert den Wert der ursprünglichen Werte nicht, es sei denn, es ist ausdrücklich der Verwendung des bitweisen Zuweisungsoperators `|=` zugewiesen:

```
int a = 5;  // 0101b  (0x05)
a |= 12;    // a = 0101b | 1101b
```

^ - bitweise XOR (exklusives ODER)

```
int a = 5;     // 0101b  (0x05)
int b = 9;     // 1001b  (0x09)
int c = a ^ b; // 1100b  (0x0C)
```

```
std::cout << "a = " << a << ", b = " << b << ", c = " << c << std::endl;
```

Ausgabe

```
a = 5, b = 9, c = 12
```

Warum

Ein bisschen `XOR` (exklusiv oder) arbeitet auf Bitebene und verwendet die folgende boolesche Wahrheitstabelle:

```
true OR true = false
true OR false = true
false OR false = false
```

Beachten Sie, dass bei einer XOR-Operation `true OR true = false` Wo wie bei den Operationen `true AND/OR true = true` , ist dies die Exklusivität der XOR-Operation.

Wenn der binäre Wert für `a` (`0101`) und der binäre Wert für `b` (`1001`) zusammen `XOR` 'sind, erhalten wir den binären Wert von `1100` :

```
int a = 0 1 0 1
int b = 1 0 0 1 ^
        ---------
int c = 1 1 0 0
```

Das bitweise XOR ändert den Wert der ursprünglichen Werte nicht, es sei denn, es ist ausdrücklich der Verwendung des bitweisen Zuweisungsoperators `^=` zugewiesen:

```
int a = 5;  // 0101b  (0x05)
a ^= 9;    // a = 0101b ^ 1001b
```

Das bitweise XOR kann auf viele Arten verwendet werden und wird häufig in Bitmaskenoperationen zur Verschlüsselung und Komprimierung verwendet.

Hinweis: Das folgende Beispiel wird häufig als Beispiel für einen schönen Trick gezeigt. Sollte aber nicht im Produktionscode verwendet werden (es gibt bessere Möglichkeiten, mit `std::swap()` dasselbe Ergebnis zu erzielen).

Sie können eine XOR-Operation auch verwenden, um zwei Variablen ohne temporäre Auslagerungen auszutauschen:

```
int a = 42;
int b = 64;

// XOR swap
a ^= b;
b ^= a;
a ^= b;

std::cout << "a = " << a << ", b = " << b << "\n";
```

Um dies zu produzieren, müssen Sie eine Prüfung hinzufügen, um sicherzustellen, dass sie verwendet werden kann.

```
void doXORSwap(int& a, int& b)
{
    // Need to add a check to make sure you are not swapping the same
    // variable with itself. Otherwise it will zero the value.
    if (&a != &b)
    {
        // XOR swap
        a ^= b;
        b ^= a;
        a ^= b;
    }
}
```

Obwohl es in der Isolation wie ein schöner Trick aussieht, ist es in echtem Code nicht nützlich. xor ist keine logische Basisoperation, sondern eine Kombination von anderen: a ^ c = ~ (a & c) & (a | c)

auch in 2015+ Compiler-Variablen können binär zugewiesen werden:

```
int cn=0b0111;
```

~ - bitweise NICHT (unäre Ergänzung)

```
unsigned char a = 234;  // 1110 1010b  (0xEA)
unsigned char b = ~a;   // 0001 0101b  (0x15)

std::cout << "a = " << static_cast<int>(a) <<
             ", b = " << static_cast<int>(b) << std::endl;
```

Ausgabe

```
a = 234, b = 21
```

Warum

Ein bisschen `NOT` (unäres Komplement) arbeitet auf Bitebene und dreht einfach jedes Bit. Wenn es eine `1` , wird es in eine `0` geändert, wenn es eine `0` , wird es in eine `1` geändert. Das bitweise NICHT hat den gleichen Effekt wie XOR, wenn ein Wert gegen den Maximalwert für einen bestimmten Typ gesetzt wird:

```
unsigned char a = 234;  // 1110 1010b  (0xEA)
unsigned char b = ~a;   // 0001 0101b  (0x15)
unsigned char c = a ^ ~0;
```

Das bitweise NICHT kann auch ein bequemer Weg sein, den Maximalwert für einen bestimmten ganzzahligen Typ zu überprüfen:

```
unsigned int i = ~0;
unsigned char c = ~0;
```

```
std::cout << "max uint = " << i << std::endl <<
             "max uchar = " << static_cast<short>(c) << std::endl;
```

Das bitweise NICHT ändert den Wert des ursprünglichen Werts nicht und hat keinen zusammengesetzten Zuweisungsoperator, so dass Sie beispielsweise kein `a ~= 10` ausführen können.

Das *bitweise* NOT (`~`) sollte nicht mit dem *logischen* NOT (`!`) Verwechselt werden; Wo ein bisschen NICHT jedes Bit umkehrt, verwendet ein logisches NICHT den gesamten Wert, um seine Operation auszuführen, dh `(!1) != (~1)`

<< - Linksverschiebung

```
int a = 1;      // 0001b
int b = a << 1; // 0010b

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Ausgabe

```
a = 1, b = 2
```

Warum

Die bitweise Verschiebung nach links verschiebt die Bits des linken Werts (`a`) um die auf der rechten Seite angegebene Zahl (`1`), wobei die niederwertigsten Bits im Wesentlichen mit 0 aufgefüllt werden, wodurch der Wert `5` (binär `0000 0101`) nach links verschoben wird 4 mal (zB `5 << 4`) ergibt den Wert `80` (binär `0101 0000`). Sie können feststellen, dass das Verschieben eines Wertes um 1 Mal nach links auch mit dem Multiplizieren des Werts mit 2 identisch ist. Beispiel:

```
int a = 7;
while (a < 200) {
    std::cout << "a = " << a << std::endl;
    a <<= 1;
}

a = 7;
while (a < 200) {
    std::cout << "a = " << a << std::endl;
    a *= 2;
}
```

Es sei jedoch darauf hingewiesen, dass die Linksverschiebungsoperation *alle* Bits einschließlich des Vorzeichenbits nach links verschiebt. Beispiel:

```
int a = 2147483647; // 0111 1111 1111 1111 1111 1111 1111 1111
int b = a << 1;     // 1111 1111 1111 1111 1111 1111 1111 1110

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Mögliche Ausgabe: `a = 2147483647, b = -2`

Während einige Compiler Ergebnisse liefern, die wie erwartet erscheinen, sollte beachtet werden, dass wenn Sie eine vorzeichenbehaftete Zahl verschieben, sodass das Vorzeichenbit betroffen ist, das Ergebnis **undefiniert ist** . Es ist auch **undefiniert,** wenn die Anzahl der Bits, um die Sie verschieben möchten, eine negative Zahl ist oder größer ist als die Anzahl der Bits, die der Typ links enthalten kann. Beispiel:

```
int a = 1;
int b = a << -1;  // undefined behavior
char c = a << 20; // undefined behavior
```

Die bitweise Verschiebung nach links ändert den Wert der ursprünglichen Werte nicht, es sei denn, sie ist speziell dem bitweisen Zuweisungsoperator `<<=` zugewiesen:

```
int a = 5;  // 0101b
a <<= 1;    // a = a << 1;
```

>> - Rechtsverschiebung

```
int a = 2;      // 0010b
int b = a >> 1; // 0001b

std::cout << "a = " << a << ", b = " << b << std::endl;
```

Ausgabe

```
a = 2, b = 1
```

Warum

Die Verschiebung des rechten Bits nach rechts verschiebt die Bits des linken Werts (`a`) um die rechts angegebene Zahl (`1`); Es sollte beachtet werden, dass die Operation einer Rechtsverschiebung zwar Standard ist, aber was mit den Bits einer Rechtsverschiebung bei einer *vorzeichenbehafteten negativen* Zahl geschieht *,* ist durch die *Implementierung definiert* und kann daher nicht als tragbar angesehen werden.

```
int a = -2;
int b = a >> 1; // the value of b will be depend on the compiler
```

Es ist auch undefiniert, wenn die Anzahl der Bits, um die Sie verschieben möchten, eine negative Zahl ist. Beispiel:

```
int a = 1;
int b = a >> -1;  // undefined behavior
```

Die bitweise Verschiebung nach rechts ändert den Wert der ursprünglichen Werte nicht, es sei denn, sie ist ausdrücklich der Verwendung des zusammengesetzten Operators für die bitweise Zuweisung `>>=` zugewiesen:

```
int a = 2;  // 0010b
```

```
a >>= 1;    // a = a >> 1;
```

Bitoperatoren online lesen: https://riptutorial.com/de/cplusplus/topic/2572/bitoperatoren

Kapitel 22: C ++ - Container

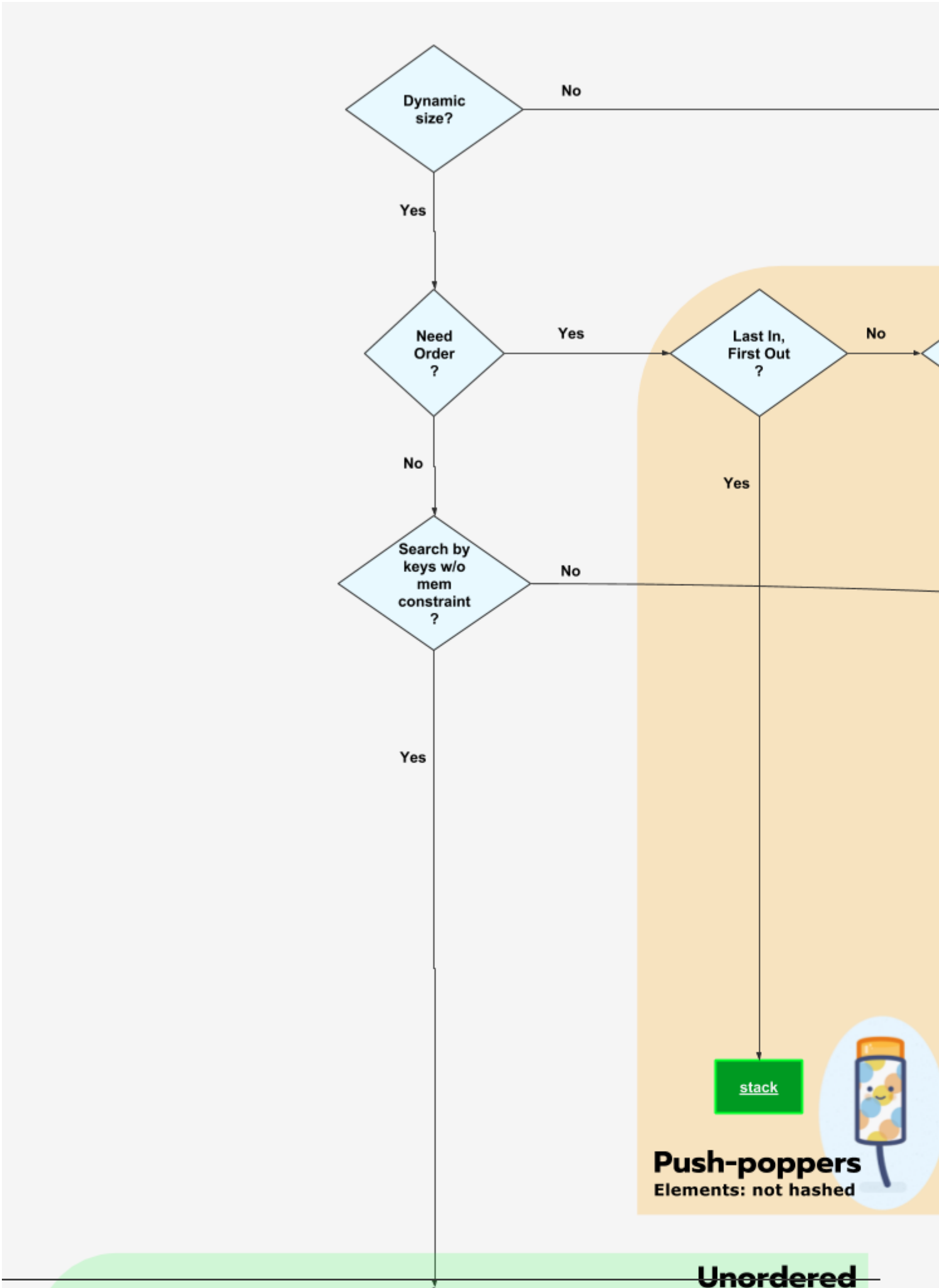
Einführung

C ++ - Container speichern eine Sammlung von Elementen. Container umfassen Vektoren, Listen, Karten usw. Mit Templates enthalten C ++ - Container Sammlungen von Grundelementen (z. B. Ints) oder benutzerdefinierten Klassen (z. B. MyClass).

Examples

C ++ Container-Flussdiagramm

Die Auswahl des zu verwendenden C ++ - Containers kann schwierig sein. Daher finden Sie hier ein einfaches Flussdiagramm, mit dem Sie entscheiden können, welcher Container für den Job geeignet ist.


Dynamic size?
No
Yes
Need Order ?
Yes
Last In, First Out ?
No
Yes
No
Search by keys w/o mem constraint ?
No
Yes
stack
Push-poppers
Elements: not hashed
Unordered
Elements: hash code

. Diese kleine Grafik im Flussdiagramm stammt von Megan Hopkins

C ++ - Container online lesen: https://riptutorial.com/de/cplusplus/topic/10848/c-plusplus---container

Kapitel 23: C ++ - Funktion "Aufruf durch Wert" vs. "Aufruf durch Referenz"

Einführung

In diesem Abschnitt werden die Unterschiede in Theorie und Implementierung erläutert, was mit den Parametern einer Funktion beim Aufruf geschieht.

Im Detail können die Parameter als Variablen vor dem Funktionsaufruf und innerhalb der Funktion betrachtet werden, wobei das sichtbare Verhalten und der Zugriff auf diese Variablen sich bei der Übergabemethode unterscheiden.

Darüber hinaus wird die Wiederverwendbarkeit von Variablen und ihrer jeweiligen Werte nach dem Funktionsaufruf ebenfalls in diesem Thema erläutert.

Examples

Aufruf nach Wert

Beim Aufruf einer Funktion werden auf dem Programmstapel neue Elemente erstellt. Dazu gehören Informationen zur Funktion sowie Platz (Speicherplätze) für die Parameter und den Rückgabewert.

Bei der Übergabe eines Parameters an eine Funktion wird der Wert der verwendeten Variablen (oder des Literal) in den Speicherplatz des Funktionsparameters kopiert. Dies bedeutet, dass jetzt zwei Speicherplätze mit demselben Wert vorhanden sind. Innerhalb der Funktion arbeiten wir nur am Parameterspeicherort.

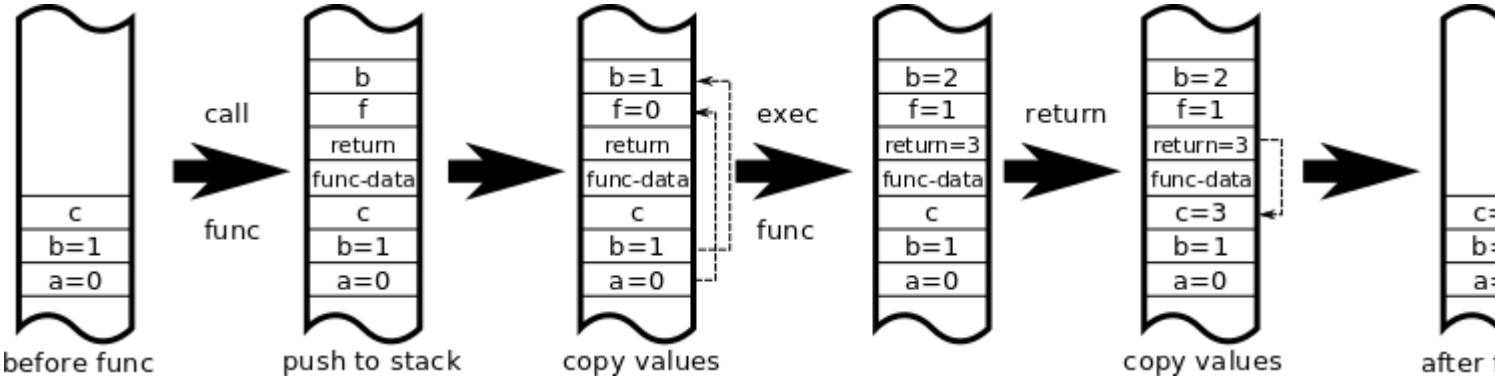
Nach dem Verlassen der Funktion wird der Speicher auf dem Programmstack entfernt (entfernt), wodurch alle Daten des Funktionsaufrufs gelöscht werden, einschließlich des Speicherorts der verwendeten Parameter. Die innerhalb der Funktion geänderten Werte wirken sich daher nicht auf die Werte der äußeren Variablen aus.

```
int func(int f, int b) {
  //new variables are created and values from the outside copied
  //f has a value of 0
  //inner_b has a value of 1
  f = 1;
  //f has a value of 1
  b = 2;
  //inner_b has a value of 2
  return f+b;
}

int main(void) {
  int a = 0;
  int b = 1; //outer_b
```

```
  int c;

  c = func(a,b);
  //the return value is copied to c

  //a has a value of 0
  //outer_b has a value of 1   <--- outer_b and inner_b are different variables
  //c has a value of 3
}
```

In diesem Code erstellen wir Variablen innerhalb der Hauptfunktion. Diese erhalten zugewiesene Werte. Beim Aufruf der Funktionen werden zwei neue Variablen erstellt: `f` und `inner_b` wobei `b` den Namen mit der äußeren Variablen teilt, nicht jedoch den Speicherort. Das Verhalten von `a<->f` und `b<->b` ist identisch.

Die folgende Grafik symbolisiert, was auf dem Stack passiert und warum sich Varibale `b` nicht ändert. Die Grafik ist nicht vollständig genau, betont jedoch das Beispiel.



Es wird "Aufruf nach Wert" genannt, da wir nicht die Variablen, sondern nur die Werte dieser Variablen übergeben.

C ++ - Funktion "Aufruf durch Wert" vs. "Aufruf durch Referenz" online lesen: https://riptutorial.com/de/cplusplus/topic/10669/c-plusplus---funktion--aufruf-durch-wert--vs---aufruf-durch-referenz-

Kapitel 24: C ++ 11-Speichermodell

Bemerkungen

Verschiedene Threads, die versuchen, auf denselben Speicherplatz zuzugreifen, nehmen an einem *Datenwettlauf teil,* wenn mindestens eine der Operationen eine Modifikation ist (auch als *Speicheroperation bezeichnet*). Diese *Datenrennen* verursachen *undefiniertes Verhalten* . Um dies zu vermeiden, muss verhindert werden, dass diese Threads gleichzeitig in Konflikt stehende Vorgänge ausführen.

Synchronisationsprimitive (Mutex, kritischer Abschnitt und dergleichen) können solche Zugriffe schützen. Das in C ++ 11 eingeführte Speichermodell definiert zwei neue tragbare Methoden zum Synchronisieren des Zugriffs auf den Speicher in einer Multithread-Umgebung: atomare Operationen und Zäune.

Atomoperationen

Es ist jetzt möglich, den angegebenen Speicherplatz mithilfe von *Atomlast*- und *Atomspeicheroperationen* zu lesen und zu schreiben. Der Einfachheit halber werden diese in die Vorlagenklasse `std::atomic<t>` . Diese Klasse wickelt einen Wert vom Typ `t` , aber dieses Mal *lädt* und *speichert* zum Objekt ist atomar.

Die Vorlage ist nicht für alle Typen verfügbar. Welche Typen verfügbar sind, ist implementierungsspezifisch, dies schließt jedoch in der Regel die meisten (oder alle) verfügbaren Integraltypen sowie Zeigertypen ein. Damit sollten `std::atomic<unsigned>` und `std::atomic<std::vector<foo> *>` verfügbar sein, während `std::atomic<std::pair<bool,char>>` höchstwahrscheinlich nicht der Fall sein wird.

Atomare Operationen haben folgende Eigenschaften:

- Alle atomaren Operationen können gleichzeitig von mehreren Threads ausgeführt werden, ohne dass es zu einem undefinierten Verhalten kommt.
- Bei einer *atomaren Last* wird entweder der Anfangswert angezeigt, mit dem das atomare Objekt erstellt wurde, oder der Wert, der über eine *atomare Speicheroperation* in dieses Objekt geschrieben wird.
- *Atomspeicher* für dasselbe Atomobjekt sind in allen Threads gleich angeordnet. Wenn ein Thread bereits den Wert einer *atomaren Speicheroperation gesehen* hat, wird bei nachfolgenden *atomaren Ladeoperationen* entweder derselbe Wert angezeigt oder der Wert, der von der nachfolgenden *atomaren Speicheroperation* gespeichert wird.
- *Atomare Lese-, Änderungs- und Schreiboperationen* ermöglichen die *atomare Last* und den *atomaren Speicher* , ohne dass ein anderer *atomarer Speicher* dazwischen liegt. Beispielsweise kann man einen Zähler aus mehreren Threads atomar inkrementieren, und unabhängig von der Konkurrenz zwischen den Threads geht kein Inkrement verloren.
- Atomare Operationen erhalten einen optionalen Parameter `std::memory_order` , der definiert,

welche zusätzlichen Eigenschaften die Operation in Bezug auf andere Speicherorte hat.

std :: memory_order	Bedeutung
`std::memory_order_relaxed`	keine zusätzlichen einschränkungen
`std::memory_order_release` → `std::memory_order_acquire`	Wenn für `load-acquire` Ladebereitschaft der von `store-release` gespeicherte Wert angezeigt wird `store-release` Filialen *sequenziert, bevor* die `store-release` , bevor die Ladevorgänge nach der *Ladebereitschaft erfolgen*
`std::memory_order_consume`	Wie `memory_order_acquire` aber nur für abhängige Ladungen
`std::memory_order_acq_rel`	kombiniert `load-acquire` und `store-release`
`std::memory_order_seq_cst`	sequentielle Konsistenz

Diese Speicherordnungs-Tags ermöglichen drei verschiedene Speicherordnungsdisziplinen: *sequenzielle Konsistenz* , *entspannt* und *Release-Acquisition* mit dem *Release-Consum von* Geschwistern.

Sequenzielle Konsistenz

Wenn für eine atomare Operation keine Speicherreihenfolge angegeben wird, wird die Reihenfolge standardmäßig auf *sequentielle Konsistenz festgelegt* . Dieser Modus kann auch explizit ausgewählt werden, indem die Operation mit `std::memory_order_seq_cst` .

Bei dieser Reihenfolge kann keine Speicheroperation die atomare Operation kreuzen. Alle Speicheroperationen, die vor der atomaren Operation sequenziert wurden, finden vor der atomaren Operation statt, und die atomare Operation findet statt vor allen Speicheroperationen, die nach dieser Sequenz ausgeführt werden. Dieser Modus ist wahrscheinlich der einfachste Grund, über den er nachdenken kann, aber er führt auch zu der größten Leistungseinschränkung. Außerdem werden alle Compiler-Optimierungen verhindert, die andernfalls versuchen würden, Operationen nach der atomaren Operation neu zu ordnen.

Entspannte Bestellung

Das Gegenteil von *sequentieller Konsistenz* ist die *entspannte* Speicheranordnung. Es wird mit dem Tag `std::memory_order_relaxed` . Durch den entspannten atomaren Betrieb werden keine anderen Speicheroperationen eingeschränkt. Der einzige Effekt, der bleibt, ist, dass die Operation selbst noch atomar ist.

Bestellung freigeben

Eine *Atomspeicheroperation* kann mit Tags versehen wird `std::memory_order_release` und ein *Atomlastbetrieb* kann mit Tags versehen wird `std::memory_order_acquire` . Die erste Operation wird als *(atomare) Speicherfreigabe bezeichnet,* während die zweite als *(atomare) Lastakquisition*

bezeichnet wird .

Bei *Last-acquire* den Wert von einem *Speicher-Release* geschrieben sieht , geschieht folgendes: alle Speicheroperationen vor dem *Laden-Release* sequenziert werden sichtbar *(geschehen vor)* Ladevorgänge , die nach dem *Last-acquire* sequenziert werden.

Atomische Lese-, Änderungs- und Schreiboperationen können auch das kumulative Tag `std::memory_order_acq_rel` . Dies macht die *Atomlast* Teil des Betriebs eine *Atom Last-acquire* , während der *Atomspeicherteil Atom-Store-Release* wird.

Der Compiler darf keine Speicheroperationen nach einer *atomaren Speicherfreigabe verschieben* . Es ist auch nicht zulässig, Ladeoperationen vor der *atomaren Lastaufnahme* (oder *Lastaufnahme*) zu verschieben.

Beachten Sie auch, dass es keine *atomare Lastfreigabe* oder *atomare Speichererfassung gibt* . Wenn Sie versuchen, solche Operationen zu erstellen, werden die Operationen *entspannt* .

Bestellung freigeben

Diese Kombination ähnelt *Release-* `std::memory_order_consume` , aber dieses Mal wird die *atomare Last* mit `std::memory_order_consume` und wird zu einer *(atomaren) load-* `std::memory_order_consume` - Operation. Dieser Modus ist derselbe wie bei der *Freigabeerfassung,* mit dem einzigen Unterschied, dass unter den nach dem *Lastverbrauch aufeinanderfolgenden Ladeoperationen* nur diese in Abhängigkeit von dem durch den *Lastverbrauch* geladenen Wert angeordnet werden.

Zäune

Zäune ermöglichen auch die Anordnung von Speicheroperationen zwischen Threads. Ein Zaun ist entweder ein Freigabezaun oder ein Zaun.

Wenn ein Freigabezaun vor einem Erfassungszaun auftritt, sind die vor dem Freigabezaun sequenzierten Speicher für Ladungen sichtbar, die nach dem Erfassungszaun sequenziert wurden. Um zu gewährleisten, dass der Freigabezaun vor dem Erfassungszaun geschieht, können andere Synchronisationsprimitive verwendet werden, einschließlich entspannter atomarer Operationen.

Examples

Notwendigkeit eines Speichermodells

```
int x, y;
bool ready = false;

void init()
{
  x = 2;
  y = 3;
  ready = true;
```

```
}
void use()
{
  if (ready)
    std::cout << x + y;
}
```

Ein Thread ruft die Funktion `init()` , während ein anderer Thread (oder Signalhandler) die Funktion `use()` aufruft. Man könnte erwarten, dass die `use()` Funktion entweder `5` druckt oder nichts tut. Dies kann aus verschiedenen Gründen nicht immer der Fall sein:

- Die CPU ordnet möglicherweise die Schreibvorgänge in `init()` so dass der tatsächlich ausgeführte Code folgendermaßen aussehen kann:

  ```
  void init()
  {
    ready = true;
    x = 2;
    y = 3;
  }
  ```

- Die CPU ordnet möglicherweise die in `use()` Lesevorgänge neu an, so dass der tatsächlich ausgeführte Code folgendermaßen wird:

  ```
  void use()
  {
    int local_x = x;
    int local_y = y;
    if (ready)
      std::cout << local_x + local_y;
  }
  ```

- Ein optimierender C ++ - Compiler kann das Programm auf ähnliche Weise neu anordnen.

Eine solche Neuordnung kann das Verhalten eines Programms, das in einem einzelnen Thread ausgeführt wird, nicht ändern, da ein Thread die Aufrufe von `init()` und `use()` nicht verschachteln kann. Auf der anderen Seite sieht ein Thread in einer Einstellung mit mehreren Threads möglicherweise einen Teil der Schreibvorgänge des anderen Threads, bei denen `use() ready==true` und garbage in `x` oder `y` oder beiden sehen kann.

Mit dem C ++ - Speichermodell kann der Programmierer angeben, welche Umordnungsvorgänge zulässig sind und welche nicht, damit ein Multithread-Programm sich auch wie erwartet verhalten kann. Das obige Beispiel kann wie folgt auf Thread-sichere Weise umgeschrieben werden:

```
int x, y;
std::atomic<bool> ready{false};

void init()
{
  x = 2;
  y = 3;
  ready.store(true, std::memory_order_release);
}
```

```
void use()
{
  if (ready.load(std::memory_order_acquire))
    std::cout << x + y;
}
```

Hier führt `init()` eine *atomare Speicherfreigabe* durch. Dadurch wird nicht nur der Wert `true` in `ready` , sondern der Compiler wird auch darüber informiert, dass er diesen Vorgang nicht vor Schreibvorgängen verschieben kann, die *zuvor sequenziert wurden* .

Die `use()` Funktion führt eine *atomare Ladeerfassungsoperation aus* . Es liest den aktuellen Wert von `ready` und verbietet dem Compiler außerdem, Leseoperationen zu platzieren, die *sequenziert werden , bevor* sie *vor* der *atomaren Ladeerfassung stattfinden* .

Diese atomaren Operationen veranlassen den Compiler außerdem dazu, alle erforderlichen Hardwarebefehle einzugeben, um die CPU darüber zu informieren, dass sie unerwünschte Neuanordnungen unterlassen muss.

Da sich die *atomare Speicherfreigabe* am selben Speicherort befindet wie die *atomare Ladeerfassung* , schreibt das Speichermodell vor, dass, wenn die *Ladeerfassungsoperation* den von der *Speicherfreigabeoperation* geschriebenen Wert sieht, alle von `init()` durchgeführten Schreibvorgänge s Faden vor dieser *Speicher-Version* wird zu Lasten sichtbar sein , die `use()` , ,s Thread führt nach seiner *last akquirieren.* Das heißt, wenn `use() ready==true` sieht, werden `x==2` und `y==3` garantiert `y==3` .

Beachten Sie, dass der Compiler und die CPU noch vor dem Schreiben in `x` nach `y` schreiben dürfen. In ähnlicher Weise können die Lesevorgänge dieser Variablen in `use()` in beliebiger Reihenfolge ausgeführt werden.

Zaun Beispiel

Das obige Beispiel kann auch mit Zäunen und entspannten atomaren Operationen implementiert werden:

```
int x, y;
std::atomic<bool> ready{false};

void init()
{
  x = 2;
  y = 3;
  atomic_thread_fence(std::memory_order_release);
  ready.store(true, std::memory_order_relaxed);
}
void use()
{
  if (ready.load(std::memory_order_relaxed))
  {
    atomic_thread_fence(std::memory_order_acquire);
    std::cout << x + y;
  }
}
```

Wenn bei der atomaren Ladeoperation der vom Atomspeicher geschriebene Wert angezeigt wird, geschieht der Speicher vor der Last und auch die Zäune: Der Freigabezaun geschieht vor dem Erfassungszaun, wodurch Schreibvorgänge nach `x` und `y` , die vor dem Freigabezaun sichtbar werden an die `std::cout` Anweisung, die dem `std::cout` folgt.

Ein Zaun kann vorteilhaft sein, wenn dadurch die Gesamtzahl der Erfassungs-, Freigabe- oder sonstigen Synchronisationsvorgänge reduziert werden kann. Zum Beispiel:

```
void block_and_use()
{
  while (!ready.load(std::memory_order_relaxed))
    ;
  atomic_thread_fence(std::memory_order_acquire);
  std::cout << x + y;
}
```

Die `block_and_use()` Funktion dreht sich, bis das `ready` Flag mit Hilfe einer entspannten `block_and_use()` gesetzt wird. Dann wird ein einzelner Erfassungszaun verwendet, um die erforderliche Speicherreihenfolge bereitzustellen.

C ++ 11-Speichermodell online lesen: https://riptutorial.com/de/cplusplus/topic/7975/c-plusplus-11-speichermodell

Kapitel 25: C ++ Streams

Bemerkungen

Der Standardkonstruktor von `std::istream_iterator` einen Iterator, der das Ende des Streams darstellt. Daher bedeutet `std::copy(std::istream_iterator<int>(ifs), std::istream_iterator<int>(), ....` das Kopieren von der aktuellen Position in `ifs` zum Ende.

Examples

String-Streams

`std::ostringstream` ist eine Klasse, deren Objekte wie ein Ausgabestrom aussehen (`std::ostringstream` Sie können sie mit dem `operator<<` schreiben), sie speichern jedoch die Schreibergebnisse und stellen sie in Form eines Streams bereit.

Betrachten Sie den folgenden Kurzcode:

```
#include <sstream>
#include <string>


using namespace std;

int main()
{
    ostringstream ss;
    ss << "the answer to everything is " << 42;
    const string result = ss.str();
}
```

Die Linie

```
ostringstream ss;
```

erstellt ein solches Objekt. Dieses Objekt wird zuerst wie ein regulärer Stream bearbeitet:

```
ss << "the answer to everything is " << 42;
```

Im Anschluss daran kann der resultierende Stream jedoch folgendermaßen erhalten werden:

```
const string result = ss.str();
```

(Das String- `result` ist gleich `"the answer to everything is 42"`).

Dies ist vor allem dann nützlich, wenn wir eine Klasse haben, für die Stream-Serialisierung definiert wurde und für die wir ein String-Formular haben möchten. Angenommen, wir haben eine

Klasse

```
class foo
{
    // All sort of stuff here.
};

ostream &operator<<(ostream &os, const foo &f);
```

Um die Zeichenfolgendarstellung eines `foo` Objekts zu erhalten,

```
foo f;
```

wir könnten gebrauchen

```
ostringstream ss;
ss << f;
const string result = ss.str();
```

Das `result` enthält dann die Zeichenfolgendarstellung des `foo` Objekts.

Lesen einer Datei bis zum Ende

Eine Textdatei zeilenweise lesen

Eine geeignete Methode, eine Textdatei Zeile für Zeile bis zum Ende zu lesen, ist in der Regel nicht aus der Dokumentation von `ifstream` . Betrachten wir einige häufige Fehler, die von Anfängern von C ++ - Programmierern begangen wurden, und eine geeignete Methode zum Lesen der Datei.

Zeilen ohne Whitespace-Zeichen

Der Einfachheit halber nehmen wir an, dass jede Zeile in der Datei keine Leerzeichen enthält.

`ifstream` hat den `operator bool()` , der true zurückgibt, wenn ein Stream fehlerfrei ist und zum Lesen bereit ist. Darüber hinaus gibt `ifstream::operator >>` einen Verweis auf den Stream selbst zurück, sodass wir EOF (sowie Fehler) in einer Zeile mit sehr eleganter Syntax lesen und überprüfen können:

```
std::ifstream ifs("1.txt");
std::string s;
while(ifs >> s) {
    std::cout << s << std::endl;
}
```

Zeilen mit Whitespace-Zeichen

`ifstream::operator >>` liest den Stream, bis ein Leerzeichen auftritt, sodass der obige Code die Wörter aus einer Zeile in separaten Zeilen druckt. Um alles bis zum Zeilenende zu lesen, verwenden Sie `std::getline` anstelle von `ifstream::operator >>` . `getline` gibt den Verweis auf den Thread zurück, mit dem er gearbeitet hat, daher ist dieselbe Syntax verfügbar:

```
while(std::getline(ifs, s)) {
    std::cout << s << std::endl;
}
```

`std::getline` sollte `std::getline` auch bis zum Ende zum Lesen einer einzeiligen Datei verwendet werden.

Eine Datei sofort in einen Puffer lesen

Schließlich lesen wir die Datei vom Anfang bis zum Ende, ohne bei einem beliebigen Zeichen, einschließlich Leerzeichen und Zeilenumbrüchen, anzuhalten. Wenn wir wissen, dass die genaue Dateigröße oder Obergrenze der Länge akzeptabel ist, können wir die Größe der Zeichenfolge ändern und dann lesen:

```
s.resize(100);
std::copy(std::istreambuf_iterator<char>(ifs), std::istreambuf_iterator<char>(),
    s.begin());
```

Andernfalls müssen wir jedes Zeichen an das Ende der Zeichenfolge einfügen. `std::back_inserter` benötigen wir `std::back_inserter` :

```
std::copy(std::istreambuf_iterator<char>(ifs), std::istreambuf_iterator<char>(),
    std::back_inserter(s));
```

Alternativ ist es möglich, eine Sammlung mit Streamdaten zu initialisieren, indem ein Konstruktor mit Iteratorbereichsargumenten verwendet wird:

```
std::vector v(std::istreambuf_iterator<char>(ifs),
    std::istreambuf_iterator<char>());
```

Beachten Sie, dass diese Beispiele auch anwendbar sind, wenn `ifs` als Binärdatei geöffnet ist:

```
std::ifstream ifs("1.txt", std::ios::binary);
```

Streams kopieren

Eine Datei kann in eine andere Datei mit Streams und Iteratoren kopiert werden:

```
std::ofstream ofs("out.file");
std::copy(std::istreambuf_iterator<char>(ifs), std::istreambuf_iterator<char>(),
    std::ostream_iterator<char>(ofs));
```

```
ofs.close();
```

oder mit einer kompatiblen Schnittstelle zu einem anderen Stream-Typ umgeleitet. Beispiel: Boost.Asio-Netzwerkstream:

```
boost::asio::ip::tcp::iostream stream;
stream.connect("example.com", "http");
std::copy(std::istreambuf_iterator<char>(ifs), std::istreambuf_iterator<char>(),
    std::ostream_iterator<char>(stream));
stream.close();
```

Arrays

Da Iteratoren als Verallgemeinerung von Zeigern betrachtet werden können, können STL-Container in den obigen Beispielen durch native Arrays ersetzt werden. So analysieren Sie Zahlen in ein Array:

```
int arr[100];
std::copy(std::istream_iterator<char>(ifs), std::istream_iterator<char>(), arr);
```

Beachten Sie den Pufferüberlauf, da die Größe der Arrays nach der Zuweisung nicht sofort geändert werden kann. Wenn der obige Code beispielsweise mit einer Datei gespeist wird, die mehr als 100 Integer-Zahlen enthält, wird versucht, außerhalb des Arrays zu schreiben und in undefiniertes Verhalten zu geraten.

Kollektionen mit Iostream drucken

Grundlegendes Drucken

`std::ostream_iterator` können Sie den Inhalt eines STL-Containers ohne explizite Schleifen in einen beliebigen Ausgabestrom drucken. Das zweite Argument des `std::ostream_iterator` Konstruktors legt das Trennzeichen fest. Zum Beispiel den folgenden Code:

```
std::vector<int> v = {1,2,3,4};
std::copy(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " ! "));
```

wird drucken

```
1 ! 2 ! 3 ! 4 !
```

Implizite Typumwandlung

`std::ostream_iterator` kann der Inhaltstyp des Containers implizit umgewandelt werden. `std::cout` wir zum Beispiel `std::cout` , um Fließkommazahlen mit 3 Stellen nach dem Dezimalpunkt zu

drucken:

```
std::cout << std::setprecision(3);
std::fixed(std::cout);
```

und instanziieren `std::ostream_iterator` mit `float` , während die enthaltenen Werte `int` bleiben:

```
std::vector<int> v = {1,2,3,4};
std::copy(v.begin(), v.end(), std::ostream_iterator<float>(std::cout, " ! "));
```

der obige Code ergibt also

```
1.000 ! 2.000 ! 3.000 ! 4.000 !
```

trotz `std::vector` gilt `int` s.

Erzeugung und Transformation

`std::generate` , `std::generate_n` und `std::transform` bieten ein sehr leistungsfähiges Werkzeug für die schnelle Datenmanipulation. Zum Beispiel mit einem Vektor:

```
std::vector<int> v = {1,2,3,4,8,16};
```

Wir können den booleschen Wert der Anweisung "x is even" für jedes Element einfach drucken:

```
std::boolalpha(std::cout); // print booleans alphabetically
std::transform(v.begin(), v.end(), std::ostream_iterator<bool>(std::cout, " "),
[](int val) {
    return (val % 2) == 0;
});
```

oder drucken Sie das quadratische Element:

```
std::transform(v.begin(), v.end(), std::ostream_iterator<int>(std::cout, " "),
[](int val) {
    return val * val;
});
```

N durch Leerzeichen getrennte Zufallszahlen drucken:

```
const int N = 10;
std::generate_n(std::ostream_iterator<int>(std::cout, " "), N, std::rand);
```

Arrays

Wie im Abschnitt zum Lesen von Textdateien können fast alle diese Überlegungen auf native

Arrays angewendet werden. Lassen Sie uns zum Beispiel quadratische Werte aus einem nativen Array drucken:

```
int v[] = {1,2,3,4,8,16};
std::transform(v, std::end(v), std::ostream_iterator<int>(std::cout, " "),
[](int val) {
    return val * val;
});
```

Dateien analysieren

Dateien in STL-Container analysieren

`istream_iterator` s ist sehr nützlich, um Zahlenfolgen oder andere analysierbare Daten in STL-Container ohne explizite Schleifen im Code zu lesen.

Explizite Containergröße verwenden:

```
std::vector<int> v(100);
std::copy(std::istream_iterator<int>(ifs), std::istream_iterator<int>(),
    v.begin());
```

oder beim Einfügen des Iterators:

```
std::vector<int> v;
std::copy(std::istream_iterator<int>(ifs), std::istream_iterator<int>(),
    std::back_inserter(v));
```

Beachten Sie, dass die Zahlen in der Eingabedatei durch eine beliebige Anzahl von Leerzeichen und Zeilenumbrüchen geteilt werden können.

Analyse heterogener Texttabellen

Da `istream::operator>>` Text bis zu einem Leerzeichen liest, kann er in `while` werden, um komplexe Datentabellen zu parsen. Wenn wir zum Beispiel eine Datei mit zwei reellen Zahlen haben, gefolgt von einer Zeichenfolge (ohne Leerzeichen) in jeder Zeile:

```
1.12 3.14 foo
2.1 2.2 barr
```

Es kann wie folgt analysiert werden:

```
std::string s;
double a, b;
while(ifs >> a >> b >> s) {
    std::cout << a << " " << b << " " << s << std::endl;
}
```

Transformation

`std::istream_iterator` Bereichen `std::istream_iterator` kann jede Bereichsmanipulationsfunktion verwendet werden. Eines davon ist `std::transform` , mit dem Daten im laufenden Betrieb verarbeitet werden können. Lassen Sie uns beispielsweise Ganzzahlwerte lesen, mit 3,14 multiplizieren und das Ergebnis in einem Gleitkommabehälter speichern:

```
std::vector<double> v(100);
std::transform(std::istream_iterator<int>(ifs), std::istream_iterator<int>(),
v.begin(),
[](int val) {
    return val * 3.14;
});
```

C ++ Streams online lesen: https://riptutorial.com/de/cplusplus/topic/7660/c-plusplus-streams

Kapitel 26: C Inkompatibilitäten

Einführung

Dies beschreibt, welcher C-Code in einem C ++ - Compiler beschädigt wird.

Examples

Reservierte Schlüsselwörter

Das erste Beispiel sind Schlüsselwörter, die in C ++ einen besonderen Zweck haben: Folgendes ist in C zulässig, jedoch nicht in C ++.

```
int class = 5
```

Diese Fehler sind leicht zu beheben: Benennen Sie die Variable einfach um.

Schwach getippte Zeiger

In C können Zeiger in ein `void*` , das eine explizite Umwandlung in C ++ erfordert. Folgendes ist in C ++ illegal, aber in C legal:

```
void* ptr;
int* intptr = ptr;
```

Das Hinzufügen einer expliziten Besetzung macht dies möglich, kann jedoch weitere Probleme verursachen.

goto oder wechseln

In C ++ können Sie Initialisierungen mit `goto` oder `switch` nicht überspringen. Folgendes gilt für C, nicht jedoch für C ++:

```
goto foo;
int skipped = 1;
foo;
```

Diese Fehler müssen möglicherweise neu entworfen werden.

C Inkompatibilitäten online lesen: https://riptutorial.com/de/cplusplus/topic/9645/c-inkompatibilitaten

Kapitel 27: Callable Objects

Einführung

Aufrufbare Objekte sind die Auflistung aller C ++ - Strukturen, die als Funktion verwendet werden können. In der Praxis können Sie dies alles an die C ++ 17 STL-Funktion invoke () übergeben oder im Konstruktor von std :: function verwendet werden. Dazu gehören: Funktionszeiger, Klassen mit operator (), Klassen mit implizit Konvertierungen, Verweise auf Funktionen, Zeiger auf Elementfunktionen, Zeiger auf Elementdaten, Lambdas. Die aufrufbaren Objekte werden in vielen STL-Algorithmen als Prädikat verwendet.

Bemerkungen

Ein sehr nützliches Gespräch von Stephan T. Lavavej (<funktional>: Neues und korrekte Verwendung) (Folien) führt zur Basis dieser Dokumentation.

Examples

Funktionszeiger

Funktionszeiger sind die grundlegendste Art, Funktionen weiterzugeben, die auch in C verwendet werden können (weitere Informationen finden Sie in der C-Dokumentation).

Für aufrufbare Objekte kann ein Funktionszeiger definiert werden als:

```
typedef returnType(*name)(arguments);                        // All
using name = returnType(*)(arguments);                       // <= C++11
using name = std::add_pointer<returnType(arguments)>::type; // <= C++11
using name = std::add_pointer_t<returnType(arguments)>;     // <= C++14
```

Wenn wir einen Funktionszeiger zum Schreiben unserer eigenen Vektorsortierung verwenden würden, würde dies folgendermaßen aussehen:

```
using LessThanFunctionPtr = std::add_pointer_t<bool(int, int)>;
void sortVectorInt(std::vector<int>&v, LessThanFunctionPtr lessThan) {
    if (v.size() < 2)
        return;
    if (v.size() == 2) {
        if (!lessThan(v.front(), v.back())) // Invoke the function pointer
            std::swap(v.front(), v.back());
        return;
    }
    std::sort(v, lessThan);
}

bool lessThanInt(int lhs, int rhs) { return lhs < rhs; }
sortVectorInt(vectorOfInt, lessThanInt); // Passes the pointer to a free function
```

```
struct GreaterThanInt {
   static bool cmp(int lhs, int rhs) { return lhs > rhs; }
};
sortVectorInt(vectorOfInt, &GreaterThanInt::cmp); // Passes the pointer to a static member
function
```

Alternativ hätten wir den Funktionszeiger auf eine der folgenden Arten aufrufen können:

- `(*lessThan)(v.front(), v.back()) // All`
- `std::invoke(lessThan, v.front(), v.back()) // <= C++17`

Klassen mit Operator () (Functors)

Jede Klasse, die den `operator()` überlädt, kann als Funktionsobjekt verwendet werden. Diese Klassen können von Hand geschrieben werden (oft als Funktoren bezeichnet) oder automatisch vom Compiler generiert, indem Lambdas ab C ++ 11 geschrieben werden.

```
struct Person {
    std::string name;
    unsigned int age;
};

// Functor which find a person by name
struct FindPersonByName {
    FindPersonByName(const std::string &name) : _name(name) {}

    // Overloaded method which will get called
    bool operator()(const Person &person) const {
         return person.name == _name;
    }
private:
    std::string _name;
};

std::vector<Person> v; // Assume this contains data
std::vector<Person>::iterator iFind =
    std::find_if(v.begin(), v.end(), FindPersonByName("Foobar"));
// ...
```

Da Funktoren eine eigene Identität haben, können sie nicht in eine Typedef-Datei eingefügt werden, und diese müssen über ein Template-Argument akzeptiert werden. Die Definition von `std::find_if` kann folgendermaßen aussehen:

```
template<typename Iterator, typename Predicate>
Iterator find_if(Iterator begin, Iterator end, Predicate &predicate) {
     for (Iterator i = begin, i != end, ++i)
         if (predicate(*i))
             return i;
     return end;
}
```

Ab C ++ 17 kann der Aufruf des Prädikats mit invoke: `std::invoke(predicate, *i)` .

Callable Objects online lesen: https://riptutorial.com/de/cplusplus/topic/6073/callable-objects

Kapitel 28: Const Korrektheit

Syntax

- class ClassOne {public: bool non_modifying_member_function () const {/ * ... * /}};
- int ClassTwo :: non_modifying_member_function () const {/ * ... * /}
- void ClassTwo :: modifying_member_function () {/ * ... * /}
- char non_param_modding_func (const ClassOne & one, const ClassTwo * zwei) {/ * ... * /}
- float parameter_modifying_function (ClassTwo & one, ClassOne * two) {/ * ... * /}
- short ClassThree :: non_modding_non_param_modding_f (const ClassOne &) const {/ * ... * /}

Bemerkungen

`const` Korrektheit ist ein sehr hilfreiches Werkzeug zur Fehlerbehebung, da der Programmierer schnell feststellen kann, welche Funktionen versehentlich Code ändern. Außerdem wird verhindert, dass ungewollte Fehler, wie z. B. die in `Const Correct Function Parameters` gezeigten, ordnungsgemäß kompiliert werden und unbemerkt bleiben.

Es ist viel einfacher, eine Klasse für `const` Korrektheit zu entwerfen, als später einer vorhandenen Klasse die `const` Korrektheit hinzuzufügen. Wenn möglich, entwerfen jede Klasse , die sein *kann* `const` korrekt , so dass es `const` richtig, sich selbst und andere die Mühe davon später modifiziert zu speichern.

Beachten Sie, dass dies bei Bedarf auch auf die `volatile` Korrektheit angewendet werden kann, mit denselben Regeln wie für die Korrektheit von `const` Diese wird jedoch viel seltener verwendet.

Referenzen:

ISO_CPP

Verkaufen Sie mich aufrichtig

C ++ Tutorial

Examples

Die Grundlagen

`const` *Korrektheit* ist die Praxis Code so zu gestalten, dass nur Code, der eine Instanz ändern *muss* in der *Lage* ist , eine Instanz zu modifizieren (dh hat Schreibzugriff), und umgekehrt, dass jeder Code, der braucht nicht eine Instanz zu ändern , ist nicht in der Lage zu tun also (hat nur lesenden Zugriff). Dadurch wird verhindert, dass die Instanz unbeabsichtigt geändert wird, wodurch Code weniger fehleranfällig wird, und es wird dokumentiert, ob der Code den Status der Instanz ändern soll oder nicht. Außerdem können Instanzen als `const` behandelt werden, wenn sie

nicht geändert werden müssen, oder als `const` definiert, wenn sie nach der Initialisierung nicht geändert werden müssen, ohne dass die Funktionalität verloren geht.

Dies geschieht durch Elementfunktionen geben `const` CV-Qualifier , und indem Zeiger / Referenzparameter `const` , außer in dem Fall , dass sie den Zugang schreiben müssen.

```
class ConstCorrectClass {
    int x;

  public:
    int getX() const { return x; } // Function is const: Doesn't modify instance.
    void setX(int i) { x = i; }    // Not const: Modifies instance.
};

// Parameter is const: Doesn't modify parameter.
int const_correct_reader(const ConstCorrectClass& c) {
    return c.getX();
}

// Parameter isn't const: Modifies parameter.
void const_correct_writer(ConstCorrectClass& c) {
    c.setX(42);
}

const ConstCorrectClass invariant; // Instance is const: Can't be modified.
ConstCorrectClass         variant; // Instance isn't const: Can be modified.

// ...

const_correct_reader(invariant); // Good.   Calling non-modifying function on const instance.
const_correct_reader(variant);   // Good.   Calling non-modifying function on modifiable
instance.

const_correct_writer(variant);   // Good.   Calling modifying function on modifiable instance.
const_correct_writer(invariant); // Error.  Calling modifying function on const instance.
```

Aufgrund der Natur der const-Korrektheit beginnt dies mit den Member-Funktionen der Klasse und arbeitet nach außen. Wenn Sie versuchen, eine Nicht- `const` Elementfunktion von einer `const` Instanz oder von einer nicht- `const` Instanz, die als `const` behandelt wird, `const` , gibt der Compiler einen Fehler aus, wenn die cv-qualifiers verloren gehen.

Richtige Klassengestaltung

In einer `const` -correct-Klasse haben alle Memberfunktionen, die den logischen Status nicht ändern, `this` cv-Qualifikation als `const this` , dass sie das Objekt nicht ändern (abgesehen von `mutable` Feldern, die auch in `const` Instanzen frei modifiziert werden können); Wenn eine `const` cv-qualifizierte Funktion eine Referenz zurückgibt, sollte diese Referenz auch `const` . Auf diese Weise können sie sowohl für konstante als auch für nicht vom CV qualifizierte Instanzen aufgerufen werden, da ein `const T*` entweder an ein `T*` oder an ein `const T*` binden kann. Auf diese Weise können Funktionen ihre übergebenen Parameter als `const` deklarieren, wenn sie nicht geändert werden müssen, ohne dass die Funktionalität verloren geht.

Ferner wird in einer `const` richtigen Klasse, alle bestanden-by-reference Funktionsparameter sein `const` richtig, wie in diskutiert `Const Correct Function Parameters` , so dass sie nur dann geändert

werden, wenn die Funktion explizit ändern *muss.*

Sehen wir uns zunächst `this` Lebenslauf-Qualifikationsmerkmale an:

```
// Assume class Field, with member function "void insert_value(int);".

class ConstIncorrect {
    Field fld;

  public:
    ConstIncorrect(Field& f); // Modifies.

    Field& getField();        // Might modify.  Also exposes member as non-const reference,
                              //  allowing indirect modification.
    void setField(Field& f);  // Modifies.

    void doSomething(int i);  // Might modify.
    void doNothing();         // Might modify.
};

ConstIncorrect::ConstIncorrect(Field& f) : fld(f) {} // Modifies.
Field& ConstIncorrect::getField() { return fld; }    // Doesn't modify.
void ConstIncorrect::setField(Field& f) { fld = f; } // Modifies.
void ConstIncorrect::doSomething(int i) {            // Modifies.
    fld.insert_value(i);
}
void ConstIncorrect::doNothing() {}                  // Doesn't modify.


class ConstCorrectCVQ {
    Field fld;

  public:
    ConstCorrectCVQ(Field& f);     // Modifies.

    const Field& getField() const; // Doesn't modify.  Exposes member as const reference,
                                   //  preventing indirect modification.
    void setField(Field& f);       // Modifies.

    void doSomething(int i);       // Modifies.
    void doNothing() const;        // Doesn't modify.
};

ConstCorrectCVQ::ConstCorrectCVQ(Field& f) : fld(f) {}
Field& ConstCorrectCVQ::getField() const { return fld; }
void ConstCorrectCVQ::setField(Field& f) { fld = f; }
void ConstCorrectCVQ::doSomething(int i) {
    fld.insert_value(i);
}
void ConstCorrectCVQ::doNothing() const  {}

// This won't work.
// No member functions can be called on const ConstIncorrect instances.
void const_correct_func(const ConstIncorrect& c) {
    Field f = c.getField();
    c.do_nothing();
}

// But this will.
// getField() and doNothing() can be called on const ConstCorrectCVQ instances.
```

```
void const_correct_func(const ConstCorrectCVQ& c) {
    Field f = c.getField();
    c.do_nothing();
}
```

Wir können dies dann mit `Const Correct Function Parameters` kombinieren, wodurch die Klasse vollständig `const` -correct ist.

```
class ConstCorrect {
    Field fld;

  public:
    ConstCorrect(const Field& f);  // Modifies instance.  Doesn't modify parameter.

    const Field& getField() const; // Doesn't modify.  Exposes member as const reference,
                                   //  preventing indirect modification.
    void setField(const Field& f); // Modifies instance.  Doesn't modify parameter.

    void doSomething(int i);       // Modifies.  Doesn't modify parameter (passed by value).
    void doNothing() const;        // Doesn't modify.
};

ConstCorrect::ConstCorrect(const Field& f) : fld(f) {}
Field& ConstCorrect::getField() const { return fld; }
void ConstCorrect::setField(const Field& f) { fld = f; }
void ConstCorrect::doSomething(int i) {
    fld.insert_value(i);
}
void ConstCorrect::doNothing() const {}
```

Dies kann auch mit Überladen basierend auf `const` kombiniert werden, in dem Fall, dass ein Verhalten gewünscht wird, wenn die Instanz `const` , und ein anderes Verhalten, wenn dies nicht der Fall ist. Eine häufige Verwendung hierfür sind Konstanten, die Zugriffsmethoden bereitstellen, die nur dann eine Änderung zulassen, wenn der Container selbst nicht `const` .

```
class ConstCorrectContainer {
    int arr[5];

  public:
    // Subscript operator provides read access if instance is const, or read/write access
    // otherwise.
          int& operator[](size_t index)       { return arr[index]; }
    const int& operator[](size_t index) const { return arr[index]; }

    // ...
};
```

Dies wird allgemein in der Standard - Bibliothek verwendet, wobei die meisten Container Überlastungen bieten nehmen `const` ness zu berücksichtigen.

Const Correct Funktionsparameter

In einer `const` -correct-Funktion werden alle Parameter, die als Referenz übergeben werden, als `const` markiert, sofern sie nicht direkt oder indirekt von der Funktion geändert werden. Dadurch

wird verhindert, dass der Programmierer versehentlich etwas ändert, das nicht geändert werden soll. Dadurch kann die Funktion sowohl nehmen `const` und nicht-cv-qualifizierte Instanzen, und wiederum bewirkt, dass die Instanz ist `this` der Typ zu sein , `const T*` wenn eine Elementfunktion aufgerufen wird , wo `T` die Klasse Typ ist.

```
struct Example {
    void func()       { std::cout << 3 << std::endl; }
    void func() const { std::cout << 5 << std::endl; }
};

void const_incorrect_function(Example& one, Example* two) {
    one.func();
    two->func();
}

void const_correct_function(const Example& one, const Example* two) {
    one.func();
    two->func();
}

int main() {
    Example a, b;
    const_incorrect_function(a, &b);
    const_correct_function(a, &b);
}

// Output:
3
3
5
5
```

Die Auswirkungen davon sind zwar weniger sichtbar als die von `const` korrekten Klassendesigns (bei `const` -correct-Funktionen und `const` -incorrect-Klassen werden Kompilierungsfehler verursacht, während `const` -correct-Klassen und `const` -incorrect-Funktionen ordnungsgemäß kompiliert werden), `const` Funktionen werden viele Fehler abfangen, durch die falsche Funktionen `const` würden, wie z. B. die unten stehende. [Beachten Sie jedoch, dass ein `const` -fehlerhafter Funktion Kompilierungsfehlern verursachen *wird* , wenn ein übergebenes `const` Beispiel , wenn es eine nicht erwartete `const` ein.]

```
// Read value from vector, then compute & return a value.
// Caches return values for speed.
template<typename T>
const T& bad_func(std::vector<T>& v, Helper<T>& h) {
    // Cache values, for future use.
    // Once a return value has been calculated, it's cached & its index is registered.
    static std::vector<T> vals = {};

    int v_ind = h.get_index();               // Current working index for v.
    int vals_ind = h.get_cache_index(v_ind); // Will be -1 if cache index isn't registered.

    if (vals.size() && (vals_ind != -1) && (vals_ind < vals.size()) && !(h.needs_recalc())) {
        return vals[h.get_cache_index(v_ind)];
    }

    T temp = v[v_ind];
```

```
    temp -= h.poll_device();
    temp *= h.obtain_random();
    temp += h.do_tedious_calculation(temp, v[h.get_last_handled_index()]);

    // We're feeling tired all of a sudden, and this happens.
    if (vals_ind != -1) {
        vals[vals_ind] = temp;
    } else {
        v.push_back(temp);  // Oops.  Should've been accessing vals.
        vals_ind = vals.size() - 1;
        h.register_index(v_ind, vals_ind);
    }

    return vals[vals_ind];
}

// Const correct version.  Is identical to above version, so most of it shall be skipped.
template<typename T>
const T& good_func(const std::vector<T>& v, Helper<T>& h) {
    // ...

    // We're feeling tired all of a sudden, and this happens.
    if (vals_ind != -1) {
        vals[vals_ind] = temp;
    } else {
        v.push_back(temp);  // Error: discards qualifiers.
        vals_ind = vals.size() - 1;
        h.register_index(v_ind, vals_ind);
    }

    return vals[vals_ind];
}
```

Const Correctness als Dokumentation

Eine der nützlicheren Dinge bei der `const` Korrektheit ist die Möglichkeit, Code zu dokumentieren und dem Programmierer und anderen Benutzern bestimmte Garantien zu geben. Diese Garantien werden vom Compiler aus `const` Konstanz erzwungen, wobei eine mangelnde `const` , dass der Code sie nicht zur Verfügung stellt.

`const` CV-Qualifizierte Mitgliederfunktionen:

- Es kann angenommen werden, dass jede Member-Funktion, die `const` ist, die Instanz lesen soll, und:
    - Ändern Sie nicht den logischen Status der Instanz, für die sie aufgerufen werden. Daher dürfen sie keine Membervariablen der Instanz ändern, für die sie aufgerufen werden, mit Ausnahme von `mutable` Variablen.
    - Ruft keine *anderen* Funktionen auf, die Mitgliedervariablen der Instanz ändern würden, mit Ausnahme von `mutable` Variablen.
- Umgekehrt kann davon ausgegangen werden, dass jede Member-Funktion, die nicht `const` ist, die Instanz ändern möchte, und:
    - Kann den logischen Zustand ändern oder nicht ändern.
    - Ruft möglicherweise andere Funktionen auf, die den logischen Zustand ändern.

Dies kann verwendet werden, um Annahmen über den Status des Objekts zu treffen, nachdem eine bestimmte Elementfunktion aufgerufen wurde, auch ohne die Definition dieser Funktion zu sehen:

```
// ConstMemberFunctions.h

class ConstMemberFunctions {
    int val;
    mutable int cache;
    mutable bool state_changed;

  public:
    // Constructor clearly changes logical state.  No assumptions necessary.
    ConstMemberFunctions(int v = 0);

    // We can assume this function doesn't change logical state, and doesn't call
    //  set_val().  It may or may not call squared_calc() or bad_func().
    int calc() const;

    // We can assume this function doesn't change logical state, and doesn't call
    //  set_val().  It may or may not call calc() or bad_func().
    int squared_calc() const;

    // We can assume this function doesn't change logical state, and doesn't call
    //  set_val().  It may or may not call calc() or squared_calc().
    void bad_func() const;

    // We can assume this function changes logical state, and may or may not call
    //  calc(), squared_calc(), or bad_func().
    void set_val(int v);
};
```

Aufgrund von `const` Regeln werden diese Annahmen tatsächlich vom Compiler durchgesetzt.

```
// ConstMemberFunctions.cpp

ConstMemberFunctions::ConstMemberFunctions(int v /* = 0*/)
  : cache(0), val(v), state_changed(true) {}

// Our assumption was correct.
int ConstMemberFunctions::calc() const {
    if (state_changed) {
        cache = 3 * val;
        state_changed = false;
    }

    return cache;
}

// Our assumption was correct.
int ConstMemberFunctions::squared_calc() const {
    return calc() * calc();
}

// Our assumption was incorrect.
// Function fails to compile, due to `this` losing qualifiers.
void ConstMemberFunctions::bad_func() const {
    set_val(863);
}
```

```
// Our assumption was correct.
void ConstMemberFunctions::set_val(int v) {
    if (v != val) {
        val = v;
        state_changed = true;
    }
}
```

`const` Funktionsparameter:

- Es kann angenommen werden, dass jede Funktion mit einem oder mehreren Parametern, die `const` sind, die Absicht hat, diese Parameter zu lesen, und:
    - Ändern Sie diese Parameter nicht und rufen Sie keine Mitgliedsfunktionen auf, die sie ändern würden.
    - Übergeben Sie diese Parameter nicht an *andere* Funktionen, die sie ändern und / oder Member-Funktionen aufrufen, die sie ändern.
- Umgekehrt kann davon ausgegangen werden, dass jede Funktion mit einem oder mehreren Parametern, die nicht `const` sind, beabsichtigt ist, diese Parameter zu ändern.
    - Kann diese Parameter ändern oder nicht ändern oder Member-Funktionen aufrufen, die sie ändern würden.
    - Diese Parameter können an andere Funktionen übergeben werden oder nicht, die sie modifizieren und / oder Member-Funktionen aufrufen, die sie modifizieren.

Dies kann verwendet werden, um Annahmen über den Zustand der Parameter zu treffen, nachdem sie an eine bestimmte Funktion übergeben wurden, auch ohne dass deren Definition definiert wird.

```
// function_parameter.h

// We can assume that c isn't modified (and c.set_val() isn't called), and isn't passed
//  to non_qualified_function_parameter().  If passed to one_const_one_not(), it is the first
//  parameter.
void const_function_parameter(const ConstMemberFunctions& c);

// We can assume that c is modified and/or c.set_val() is called, and may or may not be passed
//  to any of these functions.  If passed to one_const_one_not, it may be either parameter.
void non_qualified_function_parameter(ConstMemberFunctions& c);

// We can assume that:
  // l is not modified, and l.set_val() won't be called.
  // l may or may not be passed to const_function_parameter().
  // r is modified, and/or r.set_val() may be called.
  // r may or may not be passed to either of the preceding functions.
void one_const_one_not(const ConstMemberFunctions& l, ConstMemberFunctions& r);

// We can assume that c isn't modified (and c.set_val() isn't called), and isn't passed
//  to non_qualified_function_parameter().  If passed to one_const_one_not(), it is the first
//  parameter.
void bad_parameter(const ConstMemberFunctions& c);
```

Aufgrund von `const` Regeln werden diese Annahmen tatsächlich vom Compiler durchgesetzt.

```
// function_parameter.cpp

// Our assumption was correct.
void const_function_parameter(const ConstMemberFunctions& c) {
    std::cout << "With the current value, the output is: " << c.calc() << '\n'
              << "If squared, it's: " << c.squared_calc()
              << std::endl;
}

// Our assumption was correct.
void non_qualified_function_parameter(ConstMemberFunctions& c) {
    c.set_val(42);
    std::cout << "For the value 42, the output is: " << c.calc() << '\n'
              << "If squared, it's: " << c.squared_calc()
              << std::endl;
}

// Our assumption was correct, in the ugliest possible way.
// Note that const correctness doesn't prevent encapsulation from intentionally being broken,
//  it merely prevents code from having write access when it doesn't need it.
void one_const_one_not(const ConstMemberFunctions& l, ConstMemberFunctions& r) {
    // Let's just punch access modifiers and common sense in the face here.
    struct Machiavelli {
        int val;
        int unimportant;
        bool state_changed;
    };
    reinterpret_cast<Machiavelli&>(r).val = l.calc();
    reinterpret_cast<Machiavelli&>(r).state_changed = true;

    const_function_parameter(l);
    const_function_parameter(r);
}

// Our assumption was incorrect.
// Function fails to compile, due to `this` losing qualifiers in c.set_val().
void bad_parameter(const ConstMemberFunctions& c) {
    c.set_val(18);
}
```

Während es möglich *ist* , zu umgehen `const` Korrektheit , und durch die Erweiterung diese Garantien zu brechen, muss dies absichtlich (nur Verkapselung mit Gleichem zu brechen durch den Programmierer durchgeführt wird `Machiavelli` , und wird wahrscheinlich dazu führen , nicht definiertes Verhalten, oben).

```
class DealBreaker : public ConstMemberFunctions {
  public:
    DealBreaker(int v = 0);

    // A foreboding name, but it's const...
    void no_guarantees() const;
}

DealBreaker::DealBreaker(int v /* = 0 */) : ConstMemberFunctions(v) {}

// Our assumption was incorrect.
// const_cast removes const-ness, making the compiler think we know what we're doing.
void DealBreaker::no_guarantees() const {
```

```
    const_cast<DealBreaker*>(this)->set_val(823);
}

// ...

const DealBreaker d(50);
d.no_guarantees(); // Undefined behaviour: d really IS const, it may or may not be modified.
```

Dies ist jedoch aufgrund der Programmierer erfordert sehr speziell den Compiler zu *sagen* , dass sie beabsichtigen , zu ignorieren `const` ness und inkonsistent über Compiler ist, ist es im Allgemeinen sicher anzunehmen , dass `const` richtigen Code zu tun , so unterlassen , sofern nicht anders angegeben.

Const Korrektheit online lesen: https://riptutorial.com/de/cplusplus/topic/7217/const-korrektheit

Kapitel 29: constexpr

Einführung

`constexpr` ist ein Schlüsselwort , das verwendet werden kann, um den Wert einer Variablen als konstanten Ausdruck zu kennzeichnen, eine Funktion, die möglicherweise in konstanten Ausdrücken verwendet werden kann, oder (seit C ++ 17) eine if-Anweisung , bei der nur einer ihrer Zweige zum Kompilieren ausgewählt wurde.

Bemerkungen

Das Schlüsselwort `constexpr` wurde in C ++ 11 hinzugefügt, aber seit der Veröffentlichung des C ++ 11-Standards wurde es nicht von allen großen Compilern unterstützt. zu der Zeit, als der C ++ 11-Standard veröffentlicht wurde. Zum Zeitpunkt der Veröffentlichung von C ++ 14 unterstützen alle großen Compiler `constexpr` .

Examples

Constexpr-Variablen

Eine deklarierte `constexpr` ist implizit `const` und ihr Wert kann als konstanter Ausdruck verwendet werden.

Vergleich mit `#define`

Ein `constexpr` ist typsichere Ersatz für `#define` basierte Kompilierung-Ausdrücke. Mit `constexpr` der zur Kompilierzeit ausgewertete Ausdruck durch das Ergebnis ersetzt. Zum Beispiel:

C ++ 11

```
int main()
{
   constexpr int N = 10 + 2;
   cout << N;
}
```

erzeugt den folgenden Code:

```
cout << 12;
```

Ein vorprozessorgestütztes Kompilierungszeitmakro würde sich unterscheiden. Erwägen:

```
#define N 10 + 2

int main()
{
    cout << N;
```

```
}
```

wird herstellen:

```
cout << 10 + 2;
```

das wird offensichtlich in `cout << 10 + 2;` . Der Compiler müsste jedoch mehr Arbeit erledigen. Es wird auch ein Problem erstellt, wenn es nicht korrekt verwendet wird.

Zum Beispiel (mit `#define`):

```
cout << N * 2;
```

Formen:

```
cout << 10 + 2 * 2; // 14
```

Ein vorher ausgewertetes `constexpr` würde jedoch `24` .

Vergleich mit `const`

Eine `const` Variable ist eine **Variable,** die zum Speichern Speicher benötigt. Ein `constexpr` nicht. Ein `constexpr` erzeugt eine Kompilierungszeitkonstante, die nicht geändert werden kann. Sie können argumentieren, dass `const` auch nicht geändert werden darf. Aber bedenken Sie:

```
int main()
{
   const int size1 = 10;
   const int size2 = abs(10);

   int arr_one[size1];
   int arr_two[size2];
}
```

Bei den meisten Compilern schlägt die zweite Anweisung fehl (funktioniert beispielsweise mit GCC). Die Größe eines Arrays muss, wie Sie vielleicht wissen, ein konstanter Ausdruck sein (dh er führt zu einem Kompilierzeitwert). Der zweiten Variablen `size2` wird ein Wert zugewiesen, der zur Laufzeit festgelegt wird (obwohl Sie wissen, dass es `10` , ist dies für den Compiler keine Kompilierzeit).

Dies bedeutet, dass ein `const` eine echte Konstante für die Kompilierungszeit sein kann oder nicht. Sie können nicht garantieren oder erzwingen, dass ein bestimmter `const` Wert die Kompilierungszeit ist. Sie können `#define` , hat aber eigene Fallstricke.

Verwenden Sie dazu einfach:

C ++ 11

```
int main()
{
```

```
    constexpr int size = 10;

    int arr[size];
}
```

Ein `constexpr` Ausdruck muss zu einem Kompilierungszeitwert ausgewertet werden. Daher können Sie nicht verwenden:

C ++ 11

```
constexpr int size = abs(10);
```

Es sei denn, die Funktion (`abs`) `constexpr` selbst ein `constexpr` .

Alle `constexpr` können mit `constexpr` initialisiert `constexpr` .

C ++ 11

```
constexpr bool FailFatal = true;
constexpr float PI = 3.14f;
constexpr char* site= "StackOverflow";
```

Interessanterweise und bequemerweise können Sie auch `auto` :

C ++ 11

```
constexpr auto domain = ".COM";  // const char * const domain = ".COM"
constexpr auto PI = 3.14;        // constexpr double
```

Constexpr-Funktionen

Eine Funktion, die als `constexpr` deklariert `constexpr` ist implizit inline und Aufrufe einer solchen Funktion führen möglicherweise zu konstanten Ausdrücken. Wenn die folgende Funktion beispielsweise mit Argumenten für konstante Ausdrücke aufgerufen wird, ergibt sich auch ein konstanter Ausdruck:

C ++ 11

```
constexpr int Sum(int a, int b)
{
    return a + b;
}
```

Daher kann das Ergebnis des Funktionsaufrufs als Array-gebundenes oder Template-Argument verwendet werden oder eine `constexpr` Variable initialisieren:

C ++ 11

```
int main()
{
    constexpr int S = Sum(10,20);
```

```
    int Array[S];
    int Array2[Sum(20,30)]; // 50 array size, compile time
}
```

Wenn Sie `constexpr` aus der Rückgabetypspezifikation der Funktion entfernen, funktioniert die Zuweisung an `S` nicht, da `S` eine `constexpr` Variable ist und eine `constexpr` zur Kompilierzeit zugewiesen werden muss. In ähnlicher Weise ist auch die Größe des Arrays kein Konstantenausdruck, wenn die Funktion `Sum` nicht `constexpr` .

Interessant an den `constexpr` Funktionen ist, dass Sie sie auch wie gewöhnliche Funktionen verwenden können:

C ++ 11

```
int a = 20;
auto sum = Sum(a, abs(-20));
```

`Sum` ist jetzt keine `constexpr` Funktion, sondern wird als gewöhnliche Funktion kompiliert, wobei variable (nicht konstante) Argumente verwendet werden und ein nicht konstanter Wert zurückgegeben wird. Sie müssen nicht zwei Funktionen schreiben.

Das bedeutet auch, dass wenn Sie versuchen, einen solchen Aufruf einer Nicht-const-Variablen zuzuweisen, diese nicht kompiliert wird:

C ++ 11

```
int a = 20;
constexpr auto sum = Sum(a, abs(-20));
```

Der Grund ist einfach: `constexpr` muss nur eine Kompilierungszeitkonstante zugewiesen werden. Der obige Funktionsaufruf macht `Sum` einem nicht- `constexpr` (R-Wert ist nicht-const, aber der L-Wert deklariert sich selbst zu `constexpr`).

Die Funktion `constexpr` **muss** auch eine Konstante für die Kompilierungszeit zurückgeben. Folgendes wird nicht kompilieren:

C ++ 11

```
constexpr int Sum(int a, int b)
{
    int a1 = a;     // ERROR
    return a + b;
}
```

Weil `a1` eine Nicht-Constexpr- *Variable* ist und verhindert, dass die Funktion eine echte `constexpr` Funktion ist. So dass es `constexpr` und Zuweisen `a` auch nicht - da Wert `a` (incoming Parameter) noch noch nicht bekannt ist:

C ++ 11

```
constexpr int Sum(int a, int b)
{
   constexpr int a1 = a;     // ERROR
   ..
```

Außerdem wird Folgendes nicht kompiliert:

C ++ 11

```
constexpr int Sum(int a, int b)
{
   return abs(a) + b; // or abs(a) + abs(b)
}
```

Da `abs(a)` kein konstanter Ausdruck ist (auch `abs(10)` funktioniert nicht, da `abs` kein `constexpr int` !

Was ist damit?

C ++ 11

```
constexpr int Abs(int v)
{
    return v >= 0 ? v : -v;
}

constexpr int Sum(int a, int b)
{
    return Abs(a) + b;
}
```

Wir haben unsere eigene `Abs` Funktion entwickelt, die ein `constexpr` , und der Körper von `Abs` `constexpr` auch keine Regel. Auch an der Aufrufstelle (innerhalb von `Sum`) wird der Ausdruck zu einem `constexpr` ausgewertet. Daher ist der Aufruf von `Sum(-10, 20)` ein konstanter Ausdruck der Kompilierungszeit, der sich zu `30` ergibt.

Statische if-Anweisung

C ++ 17

Die `if constexpr` Anweisung kann verwendet werden, um Code bedingt zu kompilieren. Die Bedingung muss ein konstanter Ausdruck sein. Der nicht ausgewählte Zweig wird *verworfen.* Eine verworfene Anweisung in einer Vorlage wird nicht instanziiert. Zum Beispiel:

```
template<class T, class ... Rest>
void g(T &&p, Rest &&...rs)
{
  // ... handle p
  if constexpr (sizeof...(rs) > 0)
    g(rs...);  // never instantiated with an empty argument list
}
```

Außerdem müssen Variablen und Funktionen, die nur in verworfenen Anweisungen verwendet werden, nicht definiert werden, und verworfene `return` werden nicht für die Ableitung von

Funktionstypen verwendet.

`if constexpr` von `#ifdef` . `#ifdef` kompiliert Code bedingt, jedoch nur basierend auf Bedingungen, die zum Zeitpunkt der Vorverarbeitung ausgewertet werden können. Beispielsweise kann `#ifdef` nicht verwendet werden, um Code abhängig vom Wert eines Vorlagenparameters bedingt zu kompilieren. `if constexpr` nicht dazu verwendet werden kann, syntaktisch ungültigen Code zu verwerfen, kann dies mit `#ifdef` .

```
if constexpr(false) {
    foobar;  // error; foobar has not been declared
    std::vector<int> v("hello, world");  // error; no matching constructor
}
```

constexpr online lesen: https://riptutorial.com/de/cplusplus/topic/3899/constexpr

Kapitel 30: Datei I / O

Einführung

C ++ - Datei-E / A erfolgt über *Streams* . Die wichtigsten Abstraktionen sind:

`std::istream` zum Lesen von Text.

`std::ostream` zum Schreiben von Text.

`std::streambuf` zum Lesen oder Schreiben von Zeichen.

Formatierte Eingabe verwendet `operator>>` .

Die formatierte Ausgabe verwendet den `operator<<` .

Streams verwenden `std::locale` , z. B. für Details zur Formatierung und für die Übersetzung zwischen externen Kodierungen und der internen Kodierung.

Weitere Informationen zu Streams: <iostream> Bibliothek

Examples

Datei öffnen

Das Öffnen einer Datei erfolgt für alle 3 Dateistreams (`ifstream` , `ofstream` und `fstream`) auf dieselbe Weise.

Sie können die Datei direkt im Konstruktor öffnen:

```
std::ifstream ifs("foo.txt");  // ifstream: Opens file "foo.txt" for reading only.

std::ofstream ofs("foo.txt");  // ofstream: Opens file "foo.txt" for writing only.

std::fstream iofs("foo.txt");  // fstream:  Opens file "foo.txt" for reading and writing.
```

Alternativ können Sie die Member-Funktion des Dateistreams `open()` :

```
std::ifstream ifs;
ifs.open("bar.txt");           // ifstream: Opens file "bar.txt" for reading only.

std::ofstream ofs;
ofs.open("bar.txt");           // ofstream: Opens file "bar.txt" for writing only.

std::fstream iofs;
iofs.open("bar.txt");          // fstream:  Opens file "bar.txt" for reading and writing.
```

Sie sollten **immer** überprüfen, ob eine Datei erfolgreich geöffnet wurde (auch beim Schreiben). Zu den Fehlern können gehören: Die Datei ist nicht vorhanden, die Datei hat nicht die richtigen

Zugriffsrechte, die Datei wird bereits verwendet, Datenträgerfehler sind aufgetreten, das Laufwerk ist nicht verbunden. Die Überprüfung kann wie folgt durchgeführt werden:

```
// Try to read the file 'foo.txt'.
std::ifstream ifs("fooo.txt");  // Note the typo; the file can't be opened.

// Check if the file has been opened successfully.
if (!ifs.is_open()) {
    // The file hasn't been opened; take appropriate actions here.
    throw CustomException(ifs, "File could not be opened");
}
```

Wenn der Dateipfad umgekehrte Schrägstriche enthält (z. B. auf einem Windows-System), sollten Sie diese ordnungsgemäß deaktivieren:

```
// Open the file 'c:\folder\foo.txt' on Windows.
std::ifstream ifs("c:\\folder\\foo.txt"); // using escaped backslashes
```

C ++ 11

oder verwenden Sie Rohliteral:

```
// Open the file 'c:\folder\foo.txt' on Windows.
std::ifstream ifs(R"(c:\folder\foo.txt)"); // using raw literal
```

oder verwenden Sie stattdessen Schrägstriche:

```
// Open the file 'c:\folder\foo.txt' on Windows.
std::ifstream ifs("c:/folder/foo.txt");
```

C ++ 11

Wenn Sie unter Windows eine Datei mit Nicht-ASCII-Zeichen im Pfad öffnen möchten, können Sie ein **nicht standardmäßiges** Zeichenpfadargument verwenden:

```
// Open the file 'пример\foo.txt' on Windows.
std::ifstream ifs(LR"(пример\foo.txt)"); // using wide characters with raw literal
```

Lesen aus einer Datei

Es gibt verschiedene Möglichkeiten, Daten aus einer Datei zu lesen.

Wenn Sie wissen, wie die Daten formatiert sind, können Sie den Stream-Extraktionsoperator (`>>`) verwenden. Nehmen wir an, Sie haben eine Datei namens *foo.txt,* die die folgenden Daten enthält:

```
John Doe 25 4 6 1987
Jane Doe 15 5 24 1976
```

Dann können Sie den folgenden Code verwenden, um diese Daten aus der Datei zu lesen:

```
// Define variables.
std::ifstream is("foo.txt");
std::string firstname, lastname;
int age, bmonth, bday, byear;

// Extract firstname, lastname, age, bday month, bday day, and bday year in that order.
// Note: '>>' returns false if it reached EOF (end of file) or if the input data doesn't
// correspond to the type of the input variable (for example, the string "foo" can't be
// extracted into an 'int' variable).
while (is >> firstname >> lastname >> age >> bmonth >> bday >> byear)
    // Process the data that has been read.
```

Der Stream-Extraktionsoperator `>>` extrahiert jedes Zeichen und stoppt, wenn ein Zeichen gefunden wird, das nicht gespeichert werden kann oder wenn es ein Sonderzeichen ist:

- Bei Zeichenfolgentypen stoppt der Operator bei einem Leerzeichen ( ) oder bei einem Newline (`\n`).
- Bei Nummern stoppt der Operator bei einem Nicht-Zeichen.

Das bedeutet, dass die folgende Version der Datei *foo.txt* auch vom vorherigen Code erfolgreich gelesen werden kann:

```
John
Doe 25
4 6 1987


Jane
Doe
15 5
24
1976
```

Der Stream-Operator `>>` immer den Strom zu ihm gegeben. Daher können mehrere Operatoren miteinander verkettet werden, um Daten nacheinander lesen zu können. Ein Stream kann jedoch auch als boolescher Ausdruck verwendet werden (wie im vorherigen Code in der `while` Schleife gezeigt). Dies liegt daran, dass die Stream-Klassen einen Konvertierungsoperator für den Typ `bool` . Dieser `bool()` Operator gibt `true` , solange der Stream keine Fehler enthält. Wenn ein Stream in einen Fehlerzustand wechselt (z. B. weil keine Daten mehr extrahiert werden können), gibt der `bool()` Operator den `bool() false` . Daher wird die `while` Schleife im vorherigen Code beendet, nachdem die Eingabedatei bis zum Ende gelesen wurde.

Wenn Sie eine gesamte Datei als Zeichenfolge lesen möchten, können Sie den folgenden Code verwenden:

```
// Opens 'foo.txt'.
std::ifstream is("foo.txt");
std::string whole_file;

// Sets position to the end of the file.
is.seekg(0, std::ios::end);

// Reserves memory for the file.
whole_file.reserve(is.tellg());
```

```
// Sets position to the start of the file.
is.seekg(0, std::ios::beg);

// Sets contents of 'whole_file' to all characters in the file.
whole_file.assign(std::istreambuf_iterator<char>(is),
  std::istreambuf_iterator<char>());
```

Dieser Code reserviert Speicherplatz für die `string` um nicht benötigte Speicherzuordnungen zu reduzieren.

Wenn Sie eine Datei Zeile für Zeile lesen möchten, können Sie die Funktion `getline()` :

```
std::ifstream is("foo.txt");

// The function getline returns false if there are no more lines.
for (std::string str; std::getline(is, str);) {
    // Process the line that has been read.
}
```

Wenn Sie eine feste Anzahl von Zeichen lesen möchten, können Sie die Member-Funktion `read()` des Streams verwenden:

```
std::ifstream is("foo.txt");
char str[4];

// Read 4 characters from the file.
is.read(str, 4);
```

Nach der Ausführung eines `failbit` sollten Sie immer überprüfen, ob das Fehlerstatusflag- `failbit` gesetzt ist, da es anzeigt, ob die Operation fehlgeschlagen ist oder nicht. Dies kann durch Aufrufen der Member-Funktion `fail()` des Dateistreams erfolgen:

```
is.read(str, 4); // This operation might fail for any reason.

if (is.fail())
    // Failed to read!
```

In eine Datei schreiben

Es gibt verschiedene Möglichkeiten, in eine Datei zu schreiben. Am einfachsten ist es, einen Ausgabedatei-Stream (`ofstream`) zusammen mit dem Stream- `ofstream` (`<<`) zu verwenden:

```
std::ofstream os("foo.txt");
if(os.is_open()){
    os << "Hello World!";
}
```

Anstelle von `<<` können Sie auch die Member-Funktion `write()` der Ausgabedatei verwenden:

```
std::ofstream os("foo.txt");
if(os.is_open()){
```

```
    char data[] = "Foo";

    // Writes 3 characters from data -> "Foo".
    os.write(data, 3);
}
```

Nach dem Schreiben in einen Stream sollten Sie immer überprüfen, ob das Fehlerstatusflag `badbit` gesetzt wurde, da es anzeigt, ob die Operation fehlgeschlagen ist oder nicht. Dies kann durch Aufrufen der Member-Funktion `bad()` des Ausgabedatei-Streams erfolgen:

```
os << "Hello Badbit!"; // This operation might fail for any reason.
if (os.bad())
    // Failed to write!
```

Öffnungsmodi

Beim Erstellen eines Dateistreams können Sie einen Öffnungsmodus angeben. Ein Öffnungsmodus ist im Wesentlichen eine Einstellung, um zu steuern, wie der Stream die Datei öffnet.

(Alle Modi befinden sich im `std::ios` Namespace.)

Ein Öffnungsmodus kann dem Konstruktor eines Dateistreams oder seiner `open()` -Memberfunktion als zweiter Parameter bereitgestellt werden:

```
std::ofstream os("foo.txt", std::ios::out | std::ios::trunc);

std::ifstream is;
is.open("foo.txt", std::ios::in | std::ios::binary);
```

Es ist zu beachten, dass Sie `ios::in` oder `ios::out` wenn Sie andere Flags setzen möchten, da sie von den iostream-Mitgliedern nicht implizit gesetzt werden, obwohl sie einen korrekten Standardwert haben.

Wenn Sie keinen Öffnungsmodus angeben, werden die folgenden Standardmodi verwendet:

- `ifstream` - `in`
- `ofstream` - `out`
- `fstream` - `in` und `out`

Die Dateiöffnungsmodi, die Sie von Entwurf festlegen können, sind

Modus	Bedeutung	Zum	Beschreibung
`app`	anhängen	Ausgabe	Hängt Daten am Ende der Datei an.
`binary`	binär	Input-Output	Ein- und Ausgabe erfolgen binär.
`in`	Eingang	Eingang	Öffnet die Datei zum Lesen.

Modus	Bedeutung	Zum	Beschreibung
`out`	Ausgabe	Ausgabe	Öffnet die Datei zum Schreiben.
`trunc`	kürzen	Input-Output	Entfernt den Inhalt der Datei beim Öffnen.
`ate`	am ende	Eingang	Geht beim Öffnen an das Ende der Datei.

Hinweis: Durch das Einstellen des `binary` können die Daten genau wie sie sind gelesen / geschrieben werden. Wenn Sie diese Einstellung nicht festlegen `'\n'` das Newline `'\n'` Zeichen `'\n'` in eine plattformspezifische Zeilenende-Sequenz übersetzt.

Unter Windows lautet die Zeilenende-Sequenz beispielsweise CRLF (`"\r\n"`).
Schreiben Sie: `"\n"` => `"\r\n"`
Lesen Sie: `"\r\n"` => `"\n"`

Eine Datei schließen

Das explizite Schließen einer Datei ist in C ++ selten erforderlich, da ein Dateistream automatisch die zugehörige Datei in seinem Destruktor schließt. Sie sollten jedoch versuchen, die Lebensdauer eines Dateistream-Objekts zu begrenzen, damit der Dateikennungscode nicht länger als erforderlich geöffnet bleibt. Dies kann beispielsweise geschehen, indem alle Dateioperationen in einen eigenen Bereich (`{}`) gestellt werden:

```
std::string const prepared_data = prepare_data();
{
    // Open a file for writing.
    std::ofstream output("foo.txt");

    // Write data.
    output << prepared_data;
}  // The ofstream will go out of scope here.
   // Its destructor will take care of closing the file properly.
```

Das explizite Aufrufen von `close()` ist nur erforderlich, wenn Sie dasselbe `fstream` Objekt später erneut verwenden möchten, die Datei jedoch nicht geöffnet bleiben soll:

```
// Open the file "foo.txt" for the first time.
std::ofstream output("foo.txt");

// Get some data to write from somewhere.
std::string const prepared_data = prepare_data();

// Write data to the file "foo.txt".
output << prepared_data;

// Close the file "foo.txt".
output.close();

// Preparing data might take a long time. Therefore, we don't open the output file stream
// before we actually can write some data to it.
std::string const more_prepared_data = prepare_complex_data();
```

```
// Open the file "foo.txt" for the second time once we are ready for writing.
output.open("foo.txt");

// Write the data to the file "foo.txt".
output << more_prepared_data;

// Close the file "foo.txt" once again.
output.close();
```

Einen Strom spülen

Dateistreams werden wie viele andere Streamtypen standardmäßig gepuffert. Dies bedeutet, dass Schreibvorgänge in den Stream möglicherweise nicht dazu führen, dass die zugrunde liegende Datei sofort geändert wird. Um zu erzwingen, dass alle gepufferten Schreibvorgänge sofort ausgeführt werden, können Sie den Stream *leeren* . Sie können dies entweder direkt durch Aufrufen der Methode `flush()` oder über den Stream-Manipulator `std::flush` tun:

```
std::ofstream os("foo.txt");
os << "Hello World!" << std::flush;

char data[3] = "Foo";
os.write(data, 3);
os.flush();
```

Es gibt einen Stream-Manipulator `std::endl` , der das Schreiben einer neuen Zeile mit dem Leeren des Streams kombiniert:

```
// Both following lines do the same thing
os << "Hello World!\n" << std::flush;
os << "Hello world!" << std::endl;
```

Durch Pufferung kann die Leistung beim Schreiben in einen Stream verbessert werden. Daher sollten Anwendungen, die viel schreiben, das Flushing nicht unnötig vermeiden. Im Gegensatz dazu sollten Anwendungen bei häufigem E / A-Vorgang das häufige Leeren in Betracht ziehen, um zu verhindern, dass Daten im Stream-Objekt hängen bleiben.

Lesen einer ASCII-Datei in einen std :: string

```
std::ifstream f("file.txt");

if (f)
{
  std::stringstream buffer;
  buffer << f.rdbuf();
  f.close();

  // The content of "file.txt" is available in the string `buffer.str()`
}
```

Die `rdbuf()` -Methode gibt einen Zeiger auf eine `streambuf` , der über den `stringstream::operator<<` member `stringstream::operator<<` in den `buffer` `stringstream::operator<<` kann.

Eine andere Möglichkeit (populär in Effective STL von Scott Meyers) ist:

```
std::ifstream f("file.txt");

if (f)
{
  std::string str((std::istreambuf_iterator<char>(f)),
                  std::istreambuf_iterator<char>());

  // Operations on `str`...
}
```

Dies ist schön, da nur wenig Code erforderlich ist (und das direkte Lesen von Dateien in STL-Container und nicht nur in Strings möglich ist). Bei großen Dateien kann dies jedoch langsam sein.

ANMERKUNG : Die zusätzlichen Klammern um das erste Argument des String-Konstruktors sind wichtig, um das *ärgerliche Analyseproblem* zu vermeiden.

Zu guter Letzt:

```
std::ifstream f("file.txt");

if (f)
{
  f.seekg(0, std::ios::end);
  const auto size = f.tellg();

  std::string str(size, ' ');
  f.seekg(0);
  f.read(&str[0], size);
  f.close();

  // Operations on `str`...
}
```

Dies ist wahrscheinlich die schnellste Option (unter den drei vorgeschlagenen).

Eine Datei in einen Container lesen

Im folgenden Beispiel verwenden wir `std::string` und `operator>>` , um Elemente aus der Datei zu lesen.

```
    std::ifstream file("file3.txt");

    std::vector<std::string>  v;

    std::string s;
    while(file >> s) // keep reading until we run out
    {
        v.push_back(s);
    }
```

Im obigen Beispiel durchlaufen wir einfach die Datei und lesen jeweils einen "Eintrag" mit

`operator>>` . Der gleiche `std::istream_iterator` kann mit dem `std::istream_iterator` erzielt werden, einem Eingabe-Iterator, der jeweils ein "Element" aus dem Stream liest. Die meisten Container können auch mit zwei Iteratoren erstellt werden, um den obigen Code zu vereinfachen:

```
    std::ifstream file("file3.txt");

    std::vector<std::string>  v(std::istream_iterator<std::string>{file},
                                std::istream_iterator<std::string>{});
```

Wir können dies erweitern, um beliebige Objekttypen zu lesen, indem wir einfach das Objekt `std::istream_iterator` , das wir als Vorlagenparameter für `std::istream_iterator` lesen `std::istream_iterator` . Daher können wir das Obige einfach erweitern, um Zeilen (anstatt Wörter) wie folgt zu lesen:

```
// Unfortunately there is  no built in type that reads line using >>
// So here we build a simple helper class to do it. That will convert
// back to a string when used in string context.
struct Line
{
    // Store data here
    std::string data;
    // Convert object to string
    operator std::string const&() const {return data;}
    // Read a line from a stream.
    friend std::istream& operator>>(std::istream& stream, Line& line)
    {
        return std::getline(stream, line.data);
    }
};


    std::ifstream file("file3.txt");

    // Read the lines of a file into a container.
    std::vector<std::string>  v(std::istream_iterator<Line>{file},
                                std::istream_iterator<Line>{});
```

Lesen einer `struct` aus einer formatierten Textdatei.

C ++ 11

```
struct info_type
{
    std::string name;
    int age;
    float height;

    // we define an overload of operator>> as a friend function which
    // gives in privileged access to private data members
    friend std::istream& operator>>(std::istream& is, info_type& info)
    {
        // skip whitespace
        is >> std::ws;
        std::getline(is, info.name);
        is >> info.age;
        is >> info.height;
```

```
        return is;
    }
};

void func4()
{
    auto file = std::ifstream("file4.txt");

    std::vector<info_type> v;

    for(info_type info; file >> info;) // keep reading until we run out
    {
        // we only get here if the read succeeded
        v.push_back(info);
    }

    for(auto const& info: v)
    {
        std::cout << "  name: " << info.name << '\n';
        std::cout << "   age: " << info.age << " years" << '\n';
        std::cout << "height: " << info.height << "lbs" << '\n';
        std::cout << '\n';
    }
}
```

file4.txt

```
Wogger Wabbit
2
6.2
Bilbo Baggins
111
81.3
Mary Poppins
29
154.8
```

Ausgabe:

```
name: Wogger Wabbit
 age: 2 years
height: 6.2lbs

name: Bilbo Baggins
 age: 111 years
height: 81.3lbs

name: Mary Poppins
 age: 29 years
height: 154.8lbs
```

Datei kopieren

```
std::ifstream  src("source_filename", std::ios::binary);
std::ofstream  dst("dest_filename",   std::ios::binary);
dst << src.rdbuf();
```

C ++ 17

Mit C ++ 17 können Sie eine Datei standardmäßig mit dem Header `<filesystem>` und mit `copy_file` :

```
std::fileystem::copy_file("source_filename", "dest_filename");
```

Die Dateisystembibliothek wurde ursprünglich als `boost.filesystem` und ab C ++ 17 mit ISO C ++ zusammengeführt.

Dateiende innerhalb einer Schleifenbedingung prüfen, schlechte Praxis?

`eof` gibt `true` erst **nach** dem Ende der Datei zu lesen. Es bedeutet NICHT, dass der nächste Lesevorgang das Ende des Streams ist.

```
while (!f.eof())
{
  // Everything is OK

  f >> buffer;

  // What if *only* now the eof / fail bit is set?

  /* Use `buffer` */
}
```

Sie könnten richtig schreiben:

```
while (!f.eof())
{
  f >> buffer >> std::ws;

  if (f.fail())
    break;

  /* Use `buffer` */
}
```

aber

```
while (f >> buffer)
{
  /* Use `buffer` */
}
```

ist einfacher und weniger fehleranfällig.

Weitere referenzen:

- `std::ws` : verwirft führende Leerzeichen aus einem Eingabestrom
- `std::basic_ios::fail` : `std::basic_ios::fail` `true` zurück `true` wenn im zugeordneten Stream ein Fehler aufgetreten ist

Schreiben von Dateien mit nicht standardmäßigen Gebietsschemaeinstellungen

Wenn Sie eine Datei mit anderen Ländereinstellungen als Standard schreiben müssen, können Sie `std::locale` und `std::basic_ios::imbue()` um dies für einen bestimmten Dateistream zu tun:

Anleitung zur Verwendung:

- Sie sollten immer einen lokalen Wert auf einen Stream anwenden, bevor Sie die Datei öffnen.
- Sobald der Stream durchdrungen ist, sollten Sie das Gebietsschema nicht ändern.

Gründe für Einschränkungen: Das Erstellen eines Dateistreams mit einem Gebietsschema hat ein undefiniertes Verhalten, wenn das aktuelle Gebietsschema nicht zustandsunabhängig ist oder nicht auf den Anfang der Datei zeigt.

UTF-8-Streams (und andere) sind nicht zustandsunabhängig. Ein Dateistream mit einem UTF-8-Gebietsschema kann auch versuchen, die Stücklistenmarkierung aus der Datei zu lesen, wenn sie geöffnet wird. Wenn Sie also die Datei öffnen, werden möglicherweise Zeichen aus der Datei gelesen, und es befindet sich nicht am Anfang.

```
#include <iostream>
#include <fstream>
#include <locale>

int main()
{
  std::cout << "User-preferred locale setting is "
            << std::locale("").name().c_str() << std::endl;

  // Write a floating-point value using the user's preferred locale.
  std::ofstream ofs1;
  ofs1.imbue(std::locale(""));
  ofs1.open("file1.txt");
  ofs1 << 78123.456 << std::endl;

  // Use a specific locale (names are system-dependent)
  std::ofstream ofs2;
  ofs2.imbue(std::locale("en_US.UTF-8"));
  ofs2.open("file2.txt");
  ofs2 << 78123.456 << std::endl;

  // Switch to the classic "C" locale
  std::ofstream ofs3;
  ofs3.imbue(std::locale::classic());
  ofs3.open("file3.txt");
  ofs3 << 78123.456 << std::endl;
}
```

Das explizite Umschalten auf das klassische Gebietsschema "C" ist hilfreich, wenn Ihr Programm ein anderes Standardgebietsschema verwendet und Sie einen festen Standard für das Lesen und Schreiben von Dateien sicherstellen möchten. Bei einem bevorzugten Gebietsschema "C" schreibt das Beispiel

```
78,123.456
78,123.456
78123.456
```

Wenn das bevorzugte Gebietsschema beispielsweise Deutsch ist und daher ein anderes Zahlenformat verwendet, schreibt das Beispiel

```
78 123,456
78,123.456
78123.456
```

(Beachten Sie das Komma in der ersten Zeile).

Datei I / O online lesen: https://riptutorial.com/de/cplusplus/topic/496/datei-i---o

Kapitel 31: Datenstrukturen in C ++

Examples

Implementierung der Linked List in C ++

Erstellen eines Listenknotens

```
class listNode
{
    public:
    int data;
    listNode *next;
    listNode(int val):data(val),next(NULL){}
};
```

Listenklasse erstellen

```
class List
{
    public:
    listNode *head;
    List():head(NULL){}
    void insertAtBegin(int val);
    void insertAtEnd(int val);
    void insertAtPos(int val);
    void remove(int val);
    void print();
    ~List();
};
```

Fügen Sie einen neuen Knoten am Anfang der Liste ein

```
void List::insertAtBegin(int val)//inserting at front of list
{
    listNode *newnode = new listNode(val);
    newnode->next=this->head;
    this->head=newnode;
}
```

Fügen Sie einen neuen Knoten am Ende der Liste ein

```
void List::insertAtEnd(int val) //inserting at end of list
{
    if(head==NULL)
    {
        insertAtBegin(val);
        return;
    }
    listNode *newnode = new listNode(val);
    listNode *ptr=this->head;
    while(ptr->next!=NULL)
```

```
    {
        ptr=ptr->next;
    }
    ptr->next=newnode;
}
```

An einer bestimmten Position in der Liste einfügen

```
void List::insertAtPos(int pos,int val)
{
    listNode *newnode=new listNode(val);
    if(pos==1)
    {
        //as head
        newnode->next=this->head;
        this->head=newnode;
        return;
    }
    pos--;
    listNode *ptr=this->head;
    while(ptr!=NULL && --pos)
    {
        ptr=ptr->next;
    }
    if(ptr==NULL)
    return;//not enough elements
    newnode->next=ptr->next;
    ptr->next=newnode;
}
```

Einen Knoten aus der Liste entfernen

```
void List::remove(int toBeRemoved)//removing an element
{
    if(this->head==NULL)
    return; //empty
    if(this->head->data==toBeRemoved)
    {
        //first node to be removed
        listNode *temp=this->head;
        this->head=this->head->next;
        delete(temp);
        return;
    }
    listNode *ptr=this->head;
    while(ptr->next!=NULL && ptr->next->data!=toBeRemoved)
    ptr=ptr->next;
    if(ptr->next==NULL)
    return;//not found
    listNode *temp=ptr->next;
    ptr->next=ptr->next->next;
    delete(temp);
}
```

Drucken Sie die Liste aus

```
void List::print()//printing the list
```

```
{
    listNode *ptr=this->head;
    while(ptr!=NULL)
    {
        cout<<ptr->data<<" " ;
        ptr=ptr->next;
    }
    cout<<endl;
}
```

Zerstörer für die Liste

```
 List::~List()
{
    listNode *ptr=this->head,*next=NULL;
    while(ptr!=NULL)
    {
        next=ptr->next;
        delete(ptr);
        ptr=next;
    }
}
```

Datenstrukturen in C ++ online lesen:
https://riptutorial.com/de/cplusplus/topic/7485/datenstrukturen-in-c-plusplus

Kapitel 32: Datum und Uhrzeit mit Header

Examples

Messzeit mit

Die `system_clock` kann verwendet werden, um die Zeit zu messen, die während eines Teils der Programmausführung verstrichen ist.

c ++ 11

```
#include <iostream>
#include <chrono>
#include <thread>

int main() {
    auto start = std::chrono::system_clock::now(); // This and "end"'s type is
std::chrono::time_point
    { // The code to test
        std::this_thread::sleep_for(std::chrono::seconds(2));
    }
    auto end = std::chrono::system_clock::now();

    std::chrono::duration<double> elapsed = end - start;
    std::cout << "Elapsed time: " << elapsed.count() << "s";
}
```

In diesem Beispiel wurde `sleep_for` verwendet, um den aktiven Thread für eine in `std::chrono::seconds` gemessene Zeitdauer in den Ruhezustand zu versetzen. Der Code zwischen geschweiften Klammern kann jedoch jeder Funktionsaufruf sein, dessen Ausführung einige Zeit in Anspruch nimmt.

Finde die Anzahl der Tage zwischen zwei Terminen

Dieses Beispiel zeigt, wie Sie die Anzahl der Tage zwischen zwei Datumsangaben ermitteln können. Ein Datum wird durch Jahr / Monat / Tag des Monats und zusätzlich Stunde / Minute / Sekunde angegeben.

Das Programm berechnet die Anzahl der Tage in Jahren seit 2000.

```
#include <iostream>
#include <string>
#include <chrono>
#include <ctime>

/***
 * Creates a std::tm structure from raw date.
 *
 * \param year (must be 1900 or greater)
 * \param month months since January - [1, 12]
 * \param day day of the month - [1, 31]
```

```
 * \param minutes minutes after the hour - [0, 59]
 * \param seconds seconds after the minute - [0, 61](until C++11) / [0, 60] (since C++11)
 *
 * Based on http://en.cppreference.com/w/cpp/chrono/c/tm
 */
std::tm CreateTmStruct(int year, int month, int day, int hour, int minutes, int seconds) {
    struct tm tm_ret = {0};

    tm_ret.tm_sec = seconds;
    tm_ret.tm_min = minutes;
    tm_ret.tm_hour = hour;
    tm_ret.tm_mday = day;
    tm_ret.tm_mon = month - 1;
    tm_ret.tm_year = year - 1900;

    return tm_ret;
}

int get_days_in_year(int year) {

    using namespace std;
    using namespace std::chrono;

    // We want results to be in days
    typedef duration<int, ratio_multiply<hours::period, ratio<24> >::type> days;

    // Create start time span
    std::tm tm_start = CreateTmStruct(year, 1, 1, 0, 0, 0);
    auto tms = system_clock::from_time_t(std::mktime(&tm_start));

    // Create end time span
        std::tm tm_end =   CreateTmStruct(year + 1, 1, 1, 0, 0, 0);
    auto tme = system_clock::from_time_t(std::mktime(&tm_end));

    // Calculate time duration between those two dates
    auto diff_in_days = std::chrono::duration_cast<days>(tme - tms);

    return diff_in_days.count();
}

int main()
{
    for ( int year = 2000; year <= 2016; ++year )
        std::cout << "There are " << get_days_in_year(year) << " days in " << year << "\n";
}
```

Datum und Uhrzeit mit Header online lesen: https://riptutorial.com/de/cplusplus/topic/3936/datum-und-uhrzeit-mit--chrono--header

Kapitel 33: decltype

Einführung

Das Schlüsselwort `decltype` kann verwendet werden, um den Typ einer Variablen, Funktion oder eines Ausdrucks `decltype` .

Examples

Basisbeispiel

Dieses Beispiel zeigt nur, wie dieses Schlüsselwort verwendet werden kann.

```
int a = 10;

// Assume that type of variable 'a' is not known here, or it may
// be changed by programmer (from int to long long, for example).
// Hence we declare another variable, 'b' of the same type using
// decltype keyword.
decltype(a) b; // 'decltype(a)' evaluates to 'int'
```

Wenn sich zum Beispiel jemand ändert, geben Sie 'a' an:

```
float a=99.0f;
```

Dann wird der Variablentyp `b` jetzt automatisch `float` .

Ein anderes Beispiel

Nehmen wir an, wir haben Vektor:

```
std::vector<int> intVector;
```

Und wir wollen einen Iterator für diesen Vektor deklarieren. Eine naheliegende Idee ist die Verwendung von `auto` . Es kann jedoch erforderlich sein, nur eine Iterator-Variable zu deklarieren (und nichts zuzuweisen). Wir würden machen:

```
vector<int>::iterator iter;
```

Mit `decltype` es jedoch einfacher und weniger fehleranfällig (wenn sich der Typ von `intVector` ändert).

```
decltype(intVector)::iterator iter;
```

Alternative:

```
decltype(intVector.begin()) iter;
```

Im zweiten Beispiel wird der Rückgabewert von `begin` verwendet, um den tatsächlichen Typ zu ermitteln, nämlich `vector<int>::iterator` .

Wenn wir einen const_iterator brauchen, brauchen wir nur `cbegin` :

```
decltype(intVector.cbegin()) iter; // vector<int>::const_iterator
```

decltype online lesen: https://riptutorial.com/de/cplusplus/topic/9930/decltype

Kapitel 34: Deklaration verwenden

Einführung

Mit einer `using` Deklaration wird ein einzelner Name in den aktuellen Bereich eingefügt, der zuvor an anderer Stelle deklariert wurde.

Syntax

- using typename (*opt*) *nested-name*- spezifier *unqualifizierte-id* ;
- using :: *unqualified-id* ;

Bemerkungen

Eine *using-Deklaration* unterscheidet sich von einer using-Direktive , die den Compiler anweist, bei der Suche nach *einem* Namen in einem bestimmten Namespace zu suchen. Eine *using-Direktive* beginnt mit der `using namespace` .

Eine *using-Deklaration unterscheidet* sich auch von einer Alias-Deklaration, die einem vorhandenen Typ auf dieselbe Weise wie `typedef` einen neuen Namen gibt. Eine Alias-Deklaration enthält ein Gleichheitszeichen.

Examples

Namen einzeln aus einem Namespace importieren

Einmal `using` wird verwendet , um den Namen einzuführen `cout` aus dem Namensraum `std` in den Rahmen der `main` die `std::cout` - Objekt kann als bezeichnet wird `cout` allein.

```
#include <iostream>
int main() {
    using std::cout;
    cout << "Hello, world!\n";
}
```

Neu deklarieren von Mitgliedern aus einer Basisklasse, um das Ausblenden von Namen zu vermeiden

Wenn eine *using-Deklaration* im Klassenbereich vorkommt, darf nur ein Member einer Basisklasse neu deklariert werden. Beispielsweise ist die `using std::cout` im Klassenbereich nicht zulässig.

Oft wird der Name erneut deklariert, der sonst verborgen wäre. Im folgenden Code bezieht sich `Derived1::foo(const char*) d1.foo` nur auf `Derived1::foo(const char*)` und ein Kompilierungsfehler wird auftreten. Die Funktion `Base::foo(int)` ist ausgeblendet und wird überhaupt nicht

berücksichtigt. `d2.foo(42)` ist jedoch in Ordnung, da die *using-Deklaration* `Base::foo(int)` in die Gruppe der Entities mit Name `foo` in `Derived2` . Die Namenssuche findet dann sowohl `foo` s als auch die Überlastauflösung und wählt `Base::foo` .

```
struct Base {
    void foo(int);
};
struct Derived1 : Base {
    void foo(const char*);
};
struct Derived2 : Base {
    using Base::foo;
    void foo(const char*);
};
int main() {
    Derived1 d1;
    d1.foo(42);  // error
    Derived2 d2;
    d2.foo(42);  // OK
}
```

Erbauer erben

C ++ 11

Als Sonderfall kann eine *using-Deklaration* im Klassenbereich auf die Konstruktoren einer direkten Basisklasse verweisen. Diese Konstruktoren werden dann von der abgeleiteten Klasse *geerbt* und können zum Initialisieren der abgeleiteten Klasse verwendet werden.

```
struct Base {
    Base(int x, const char* s);
};
struct Derived1 : Base {
    Derived1(int x, const char* s) : Base(x, s) {}
};
struct Derived2 : Base {
    using Base::Base;
};
int main() {
    Derived1 d1(42, "Hello, world");
    Derived2 d2(42, "Hello, world");
}
```

Im obigen Code verfügen sowohl `Derived1` als auch `Derived2` über Konstruktoren, die die Argumente direkt an den entsprechenden Konstruktor von `Base` weiterleiten. `Derived1` führt die Weiterleitung explizit aus, während `Derived2` mit der C ++ 11-Funktion zum Erben von Konstruktoren implizit erfolgt.

Deklaration verwenden online lesen: https://riptutorial.com/de/cplusplus/topic/9301/deklaration-verwenden

Kapitel 35: Der ISO-C ++ - Standard

Einführung

1998 wurde der Standard erstmals veröffentlicht, wodurch C ++ eine intern standardisierte Sprache wurde. Seitdem hat sich C ++ entwickelt, was zu unterschiedlichen Dialekten von C ++ führte. Auf dieser Seite finden Sie eine Übersicht aller verschiedenen Standards und deren Änderungen gegenüber der Vorgängerversion. Die Details zur Verwendung dieser Funktionen werden auf spezielleren Seiten beschrieben.

Bemerkungen

Wenn C ++ erwähnt wird, wird häufig auf "den Standard" verwiesen. Aber was ist genau dieser Standard?

C ++ hat eine lange Geschichte. Begonnen als kleines Projekt von Bjarne Stroustrup in den Bell Labs, war es Anfang der 90er Jahre sehr beliebt. Mehrere Unternehmen erstellten C ++ - Compiler, sodass Benutzer ihre C ++ - Compiler auf einer Vielzahl von Computern ausführen können. Um dies zu erleichtern, sollten alle konkurrierenden Compiler eine einzige Definition der Sprache haben.

Zu diesem Zeitpunkt war die C-Sprache erfolgreich standardisiert worden. Dies bedeutet, dass eine formale Beschreibung der Sprache verfasst wurde. Dies wurde dem American National Standards Institute (ANSI) vorgelegt, das das Dokument zur Überprüfung öffnete und 1989 veröffentlichte. Ein Jahr später die International Organization for Standards (Da es verschiedene Akronyme in verschiedenen Sprachen hätte, wählten sie eine Form , ISO, abgeleitet von den griechischen isos (gleichwertig), übernahm den amerikanischen Standard als internationalen Standard.

Für C ++ war von Anfang an klar, dass ein internationales Interesse besteht. Eine Arbeitsgruppe innerhalb der ISO wurde gegründet (bekannt als WG21, innerhalb des Unterausschusses 22). Diese Arbeitsgruppe entwarf um 1995 einen ersten Standard. Aber wie wir Programmierer wissen, ist nichts für eine geplante Bereitstellung gefährlicher als die Last-Minute-Funktionen, und das geschah auch bei C ++. 1995 erschien eine coole neue Bibliothek mit dem Namen STL, und die in WG21 tätigen Personen beschlossen, den C ++ - Entwurfsstandard um eine abgespeckte Version zu erweitern. Natürlich wurden dadurch die Fristen nicht eingehalten und nur drei Jahre später wurde das Dokument endgültig. ISO ist eine sehr formale Organisation, daher wurde der C ++ - Standard mit dem nicht sehr marktfähigen Namen ISO / IEC 14882 getauft. Da Standards aktualisiert werden können, wurde diese genaue Version als 14882: 1998 bekannt.

Und in der Tat bestand die Forderung, den Standard zu aktualisieren. Der Standard ist ein sehr umfangreiches Dokument, das genau beschreibt, wie C ++ - Compiler arbeiten sollen. Selbst eine geringfügige Unklarheit kann es wert sein, korrigiert zu werden. Daher wurde bis 2003 ein Update unter der Nummer 14882: 2003 veröffentlicht. Dies fügte jedoch kein Feature zu C ++ hinzu; Die neuen Funktionen waren für das zweite Update geplant.

Informell wurde dieses zweite Update als C ++ 0x bezeichnet, da nicht bekannt war, ob dies bis 2008 oder 2009 dauern würde. Nun, diese Version wurde ebenfalls etwas verzögert, weshalb sie 14882: 2011 wurde.

Glücklicherweise entschied sich die WG21, dies nicht noch einmal passieren zu lassen. C ++ 11 wurde gut angenommen und ließ ein erneutes Interesse an C ++ erkennen. Um die Dynamik aufrechtzuerhalten, wurde das dritte Update innerhalb von 3 Jahren von der Planung zur Veröffentlichung auf 14882: 2014 geändert.

Die Arbeit hörte auch dort nicht auf. Der C ++ 17-Standard wurde vorgeschlagen und die Arbeit für C ++ 20 wurde gestartet.

Examples

Aktuelle Arbeitsentwürfe

Alle veröffentlichten ISO-Standards sind im ISO-Store (http://www.iso.org) erhältlich. Die Arbeitsentwürfe der C ++ - Standards sind jedoch kostenlos öffentlich verfügbar.

Die verschiedenen Versionen des Standards:

- In Kürze verfügbar (manchmal als C ++ 20 oder C ++ 2a bezeichnet): Aktueller Arbeitsentwurf (HTML-Version)
- Vorgeschlagen (manchmal als C ++ 17 oder C ++ 1z bezeichnet): März 2017 Arbeitsentwurf N4659 .
- C ++ 14 (manchmal als C ++ 1y bezeichnet): Arbeitsentwurf N4296 vom November 2014
- C ++ 11 (manchmal als C ++ 0x bezeichnet): Arbeitsentwurf für Februar 2011 N3242
- C ++ 03
- C ++ 98

C ++ 11

Der C ++ 11-Standard ist eine wichtige Erweiterung des C ++ - Standards. Nachfolgend finden Sie eine Übersicht der Änderungen, die in den isocpp-FAQs mit Links zu detaillierterer Dokumentation zusammengefasst wurden.

Spracherweiterungen

Allgemeine Merkmale

- Auto
- decltype
- Range-for-Anweisung
- Initialisierungslisten
- Einheitliche Initialisierungssyntax und -semantik

- Referenzwerte und Verschiebungssemantik
- Lambdas
- noexcept , um die Ausbreitung von Ausnahmen zu verhindern
- constexpr
- nullptr - ein Nullzeigerliteral
- Ausnahmen kopieren und erneut werfen
- Inline-Namespaces
- Benutzerdefinierte Literale

Klassen

- = default und = löschen
- Kontrolle der Standardeinstellungen zum Verschieben und Kopieren
- Konstruktoren delegieren
- Initialisierer innerhalb der Klasse
- Geerbte Konstruktoren
- Steuerelemente überschreiben: überschreiben
- Steuerelemente überschreiben: final
- Explizite Konvertierungsoperatoren

Andere Arten

- Aufzählungsklasse
- long long - eine längere ganze Zahl
- Erweiterte Ganzzahltypen
- Generalisierte Gewerkschaften
- Generalisierte PODs

Vorlagen

- Externe Vorlagen
- Template-Aliase
- Variadische Vorlagen
- Lokale Typen als Vorlagenargumente

Parallelität

- Gleichzeitiges Speichermodell
- Dynamische Initialisierung und Zerstörung mit Parallelität
- Thread-lokaler Speicher

Verschiedene Sprachfunktionen

- Was ist der Wert von __cplusplus für C ++ 11?

- Suffix Rückgabetypsyntax
- Verengung verhindern
- Rechtwinklige Klammern
- Kompatibilitäts-Assertions von static_assert
- Rohe String-Literale
- Attribute
- Ausrichtung
- C99-Funktionen

Bibliothekserweiterungen

Allgemeines

- unique_ptr
- shared_ptr
- schwach_ptr
- Müllabfuhr ABI
- Tupel
- Eigenschaften eingeben
- Funktion und Bindung
- Reguläre Ausdrücke
- Zeit-Dienstprogramme
- Zufallszahlengenerierung
- Aufteiler für Bereiche

Container und Algorithmen

- Algorithmenverbesserungen
- Containerverbesserungen
- unordered_ * container
- std :: array
- forward_list

Parallelität

- Fäden
- Gegenseitiger Ausschluss
- Schlösser
- Bedingungsvariablen
- Atomik
- Futures und Versprechen
- asynchron
- Einen Prozess aufgeben

C ++ 14

Der C ++ 14-Standard wird häufig als Bugfix für C ++ 11 bezeichnet. Es enthält nur eine begrenzte Liste von Änderungen, von denen die meisten Erweiterungen der neuen Features in C ++ 11 sind. Nachfolgend finden Sie eine Übersicht der Änderungen, die in den isocpp-FAQs mit Links zu detaillierterer Dokumentation zusammengefasst wurden.

Spracherweiterungen

- Binäre Literale
- Generalisierter Rücknahme-Abzug
- decltype (auto)
- Generalisierte Lambda-Fänge
- Generische Lambdas
- Variable Vorlagen
- Erweiterte `constexpr`
- Das Attribut `[[deprecated]]`
- Zifferntrennzeichen

Bibliothekserweiterungen

- Gemeinsames Sperren
- Benutzerdefinierte Literale für `std::` types
- `std::make_unique`
- `_t` Aliase der Transformation `_t`
- Adressierung von Tupeln nach Typ (z. B. `get<string>(t)`)
- Transparente Operatorfunktionalitäten (z. B. `greater<>(x)`)
- `std::quoted`

Veraltet / Entfernt

- `std::gets` wurde in C ++ 11 veraltet und aus C ++ 14 entfernt
- `std::random_shuffle` ist veraltet

C ++ 17

Der C ++ 17-Standard ist vollständig und wurde zur Standardisierung vorgeschlagen. In Compilern mit experimenteller Unterstützung für diese Features wird es normalerweise als C ++ 1z bezeichnet.

Spracherweiterungen

- Ausdrücke falten
- Nicht-Typ-Vorlagenargumente mit `auto` deklarieren
- Garantierte Kopiereinstellung
- Vorlagenparameterabzug für Konstruktoren
- Strukturierte Bindungen
- Kompakte verschachtelte Namespaces
- Neue Attribute: `[[fallthrough]]` , `[[nodiscard]]` , `[[maybe_unused]]`
- Standardnachricht für `static_assert`
- Initialisierer in `if` und `switch`
- Inline-Variablen
- `if constexpr`
- Reihenfolge der Ausdrucksgarantien
- Dynamische Speicherzuordnung für übergeordnete Daten

Bibliothekserweiterungen

- `std::optional`
- `std::variant`
- `std::string_view`
- `merge()` und `extract()` für assoziative Container
- Eine Dateisystembibliothek mit dem Header `<filesystem>` .
- Parallele Versionen der meisten Standardalgorithmen (im Header `<algorithm>`).
- Hinzufügen mathematischer Sonderfunktionen in der `<cmath>` .
- Knoten zwischen map <>, unordered_map <>, set <> und unordered_set <> verschieben

C ++ 03

Der C ++ 03-Standard behandelt hauptsächlich Fehlerberichte des C ++ 98-Standards. Abgesehen von diesen Mängeln wird nur eine neue Funktion hinzugefügt.

Spracherweiterungen

- Wert Initialisierung

C ++ 98

C ++ 98 ist die erste standardisierte Version von C ++. Da es als Erweiterung zu C entwickelt wurde, werden viele Features hinzugefügt, die C ++ von C unterscheiden.

Spracherweiterungen (in Bezug auf C89 / C90)

- Klassen, abgeleitete Klassen, virtuelle Memberfunktionen, const Memberfunktionen

- Funktionsüberlastung, Operatorüberlastung
- Einzeilige Kommentare (Wurde in der C-Sprache mit C99-Standard eingeführt)
- Verweise
- neu und löschen
- boolescher Typ (Wurde in der C-Sprache mit C99-Standard eingeführt)
- Vorlagen
- Namespaces
- Ausnahmen
- bestimmte Besetzungen

Bibliothekserweiterungen

- Die Standardvorlagenbibliothek

C ++ 20

C ++ 20 ist der kommende Standard von C ++, der sich derzeit in der Entwicklung befindet und auf dem C ++ 17-Standard basiert. Der Fortschritt kann auf der offiziellen ISO-CPP-Website verfolgt werden .

Die folgenden Funktionen sind einfach das, was für die nächste Version des C ++ - Standards, die für 2020 vorgesehen ist, akzeptiert wurde.

Spracherweiterungen

Bisher wurden keine Spracherweiterungen akzeptiert.

Bibliothekserweiterungen

Bisher wurden keine Bibliothekserweiterungen akzeptiert.

Der ISO-C ++ - Standard online lesen: https://riptutorial.com/de/cplusplus/topic/2742/der-iso-c-plusplus---standard

Kapitel 36: Der This Pointer

Bemerkungen

`this` Zeiger ist ein Schlüsselwort für C ++, daher ist keine Bibliothek erforderlich, um dies zu implementieren. Und vergiss nicht, `this` ist ein Zeiger! Du kannst also nicht:

```
 this.someMember();
```

Wenn Sie mit dem Pfeilsymbol auf Elementfunktionen oder Elementvariablen zugreifen, verwenden Sie das Pfeilsymbol `->` :

```
this->someMember();
```

Weitere hilfreiche Links zum besseren Verständnis `this` Zeigers:

Was ist der "Dies" -Zeiger?

http://www.geeksforgeeks.org/this-pointer-in-c/

https://www.tutorialspoint.com/cplusplus/cpp_this_pointer.htm

Examples

dieser Zeiger

Alle nicht statischen Memberfunktionen haben einen ausgeblendeten Parameter, einen Zeiger auf eine Instanz der Klasse mit dem Namen `this` . Dieser Parameter wird am Anfang der Parameterliste automatisch eingefügt und vollständig vom Compiler behandelt. Wenn innerhalb einer Member-Funktion auf ein Member der Klasse zugegriffen wird, wird durch `this` Funktion im Hintergrund darauf zugegriffen. Dadurch kann der Compiler eine einzige nicht statische Memberfunktion für alle Instanzen verwenden, und eine Memberfunktion kann andere Memberfunktionen polymorph aufrufen.

```
struct ThisPointer {
    int i;

    ThisPointer(int ii);

    virtual void func();

    int  get_i() const;
    void set_i(int ii);
};
ThisPointer::ThisPointer(int ii) : i(ii) {}
// Compiler rewrites as:
ThisPointer::ThisPointer(int ii) : this->i(ii) {}
// Constructor is responsible for turning allocated memory into 'this'.
```

```
// As the constructor is responsible for creating the object, 'this' will not be "fully"
// valid until the instance is fully constructed.

/* virtual */ void ThisPointer::func() {
    if (some_external_condition) {
        set_i(182);
    } else {
        i = 218;
    }
}
// Compiler rewrites as:
/* virtual */ void ThisPointer::func(ThisPointer* this) {
    if (some_external_condition) {
        this->set_i(182);
    } else {
        this->i = 218;
    }
}

int  ThisPointer::get_i() const { return i; }
// Compiler rewrites as:
int  ThisPointer::get_i(const ThisPointer* this) { return this->i; }

void ThisPointer::set_i(int ii) { i = ii; }
// Compiler rewrites as:
void ThisPointer::set_i(ThisPointer* this, int ii) { this->i = ii; }
```

In einem Konstruktor kann `this` sicher verwendet werden, um (implizit oder explizit) auf ein Feld zuzugreifen, das bereits initialisiert wurde, oder auf ein Feld in einer übergeordneten Klasse. umgekehrt ist (implizit oder explizit) der Zugriff auf Felder, die noch nicht initialisiert wurden, oder auf Felder in einer abgeleiteten Klasse, unsicher (da die abgeleitete Klasse noch nicht erstellt wurde und ihre Felder daher weder initialisiert noch vorhanden sind). Es ist auch nicht sicher virtuelle Mitgliederfunktionen bis hin zu nennen `this` im Konstruktor, wie alle abgeleiteten Klasse Funktionen werden nicht berücksichtigt (aufgrund der abgeleiteten Klasse noch nicht aufgebaut ist, und somit der Konstruktor noch nicht die V - Tabelle zu aktualisieren).

Beachten Sie außerdem, dass der Typ des Objekts in einem Konstruktor der Typ ist, den dieser Konstruktor erstellt. Dies gilt auch dann, wenn das Objekt als abgeleiteter Typ deklariert ist. In dem folgenden Beispiel sind `ctd_good` und `ctd_bad` beispielsweise `CtorThisBase` in `CtorThisBase()` und `CtorThis` in `CtorThis()` , obwohl der kanonische Typ `CtorThisDerived` . Da die stärker abgeleiteten Klassen um die Basisklasse herum aufgebaut werden, durchläuft die Instanz nach und nach die Klassenhierarchie, bis es sich um eine vollständig konstruierte Instanz des beabsichtigten Typs handelt.

```
class CtorThisBase {
    short s;

  public:
    CtorThisBase() : s(516) {}
};

class CtorThis : public CtorThisBase {
    int i, j, k;
```

```
  public:
    // Good constructor.
    CtorThis() : i(s + 42), j(this->i), k(j) {}

    // Bad constructor.
    CtorThis(int ii) : i(ii), j(this->k), k(b ? 51 : -51) {
        virt_func();
    }

    virtual void virt_func() { i += 2; }
};

class CtorThisDerived : public CtorThis {
    bool b;

  public:
    CtorThisDerived()       : b(true) {}
    CtorThisDerived(int ii) : CtorThis(ii), b(false) {}

    void virt_func() override { k += (2 * i); }
};

// ...

CtorThisDerived ctd_good;
CtorThisDerived ctd_bad(3);
```

Mit diesen Klassen und Memberfunktionen:

- Im guten Konstruktor für `ctd_good` :
  - `CtorThisBase` ist vollständig erstellt, wenn der `CtorThis` Konstruktor eingegeben wird. Daher befindet sich `s` während der Initialisierung von `i` in einem gültigen Zustand und kann somit aufgerufen werden.
  - `i` wird initialisiert, bevor `j(this->i)` erreicht ist. Daher befindet sich `i` während der Initialisierung von `j` in einem gültigen Zustand und kann somit darauf zugegriffen werden.
  - `j` wird initialisiert, bevor `k(j)` erreicht wird. Daher befindet sich `j` während der Initialisierung von `k` in einem gültigen Zustand und kann somit darauf zugegriffen werden.
- Im schlechten Konstruktor für `ctd_bad` :
  - `k` wird initialisiert, nachdem `j(this->k)` erreicht ist. Daher ist `k` während der Initialisierung von `j` in einem ungültigen Zustand, und der Zugriff darauf verursacht ein undefiniertes Verhalten.
  - `CtorThisDerived` wird erst nach `CtorThis` . Daher ist `b` während der Initialisierung von `k` in einem ungültigen Zustand, und der Zugriff darauf verursacht ein undefiniertes Verhalten.
  - Das Objekt `ctd_bad` ist noch ein `CtorThis` bis es `CtorThis()` verlässt, und wird nicht aktualisiert, um die `CtorThisDerived` CtorThisDerived bis `CtorThisDerived()` . Daher `virt_func()` `CtorThis::virt_func()` , unabhängig davon, ob dies oder `CtorThisDerived::virt_func()` .

Verwenden dieses Zeigers für den Zugriff auf Mitgliedsdaten

In diesem Zusammenhang ist die Verwendung des Zeigers ' `this` nicht unbedingt erforderlich, macht den Code jedoch für den Leser klarer, indem er anzeigt, dass eine bestimmte Funktion oder Variable eine Klasse ist. Ein Beispiel in dieser Situation:

```
// Example for this pointer
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Class
{
  public:
    Class();
    ~Class();
    int getPrivateNumber () const;
  private:
    int private_number = 42;
};

Class::Class(){}
Class::~Class(){}

int Class::getPrivateNumber() const
{
    return this->private_number;
}

int main()
{
    Class class_example;
    cout << class_example.getPrivateNumber() << endl;
}
```

Sehen sie in Aktion hier .

Verwenden dieses Zeigers zur Unterscheidung zwischen Mitgliedsdaten und Parametern

Dies ist eine nützliche Strategie, um Elementdaten von Parametern zu unterscheiden. Nehmen wir folgendes Beispiel:

```
// Dog Class Example
#include <iostream>
#include <string>

using std::cout;
using std::endl;

/*
* @class Dog
*   @member name
*       Dog's name
*   @function bark
*       Dog Barks!
*   @function getName
```

```
*       To Get Private
*       Name Variable
*/
class Dog
{
 public:
    Dog(std::string name);
    ~Dog();
    void  bark() const;
    std::string  getName() const;
 private:
    std::string name;
};

Dog::Dog(std::string name)
{
    /*
    *  this->name is the
    *  name variable from
    *  the class dog . and
    *  name is from the
    *  parameter of the function
    */
    this->name = name;
}

Dog::~Dog(){}

void Dog::bark() const
{
  cout << "BARK" << endl;
}

std::string  Dog::getName() const
{
    return this->name;
}


int main()
{
    Dog dog("Max");
    cout << dog.getName() << endl;
    dog.bark();
}
```

Sie können hier im Konstruktor sehen, dass wir Folgendes ausführen:

```
this->name = name;
```

Hier sehen Sie, dass wir den Parameternamen dem Namen der privaten Variablen der Klasse Dog (this-> name) zuordnen.

So sehen Sie die Ausgabe des obigen Codes: http://cpp.sh/75r7

diese Pointer CV-Qualifiers

`this` kann auch cv-qualifiziert sein, genauso wie jeder andere Zeiger. Da `this` Parameter jedoch nicht in der Parameterliste aufgeführt ist, ist hierfür eine spezielle Syntax erforderlich. Die CV-Qualifikationsmerkmale werden nach der Parameterliste aufgeführt, jedoch vor dem Hauptteil der Funktion.

```
struct ThisCVQ {
    void no_qualifier()                {} // "this" is: ThisCVQ*
    void  c_qualifier() const          {} // "this" is: const ThisCVQ*
    void  v_qualifier() volatile       {} // "this" is: volatile ThisCVQ*
    void cv_qualifier() const volatile {} // "this" is: const volatile ThisCVQ*
};
```

Da `this` um einen Parameter handelt, kann eine Funktion auf der Grundlage `this` cv-Qualifier (s) überladen werden .

```
struct CVOverload {
    int func()                { return    3; }
    int func() const          { return   33; }
    int func() volatile       { return  333; }
    int func() const volatile { return 3333; }
};
```

Wenn `this const` (einschließlich `const volatile`), kann die Funktion weder implizit noch explizit in Member-Variablen schreiben. Die einzige Ausnahme hiervon sind `mutable` Membervariablen , die unabhängig von der Konstante geschrieben werden können. Aus diesem Grund wird mit `const` angezeigt, dass die Member-Funktion den logischen Zustand des Objekts (die Art, wie das Objekt der Außenwelt erscheint) nicht ändert, selbst wenn es den physischen Zustand (die Art, wie das Objekt unter der Haube aussieht) ändert).

> Der logische Zustand ist die Art und Weise, wie das Objekt außerhalb der Beobachter erscheint. Es ist nicht direkt an den physischen Zustand gebunden und wird möglicherweise nicht einmal als physischer Zustand gespeichert. Solange externe Beobachter keine Änderungen sehen können, ist der logische Zustand konstant, selbst wenn Sie jedes einzelne Bit im Objekt umdrehen.
>
> Der physikalische Zustand, auch als bitweiser Zustand bezeichnet, ist, wie das Objekt im Speicher abgelegt wird. Dies ist das Kleinste des Objekts, die rohen Einsen und Nullen, aus denen seine Daten bestehen. Ein Objekt ist nur physisch konstant, wenn sich seine Darstellung im Speicher niemals ändert.

Beachten Sie, dass C ++ Basen `const` auf logischen Zustand ness, nicht physischen Zustand.

```
class DoSomethingComplexAndOrExpensive {
    mutable ResultType cached_result;
    mutable bool state_changed;

    ResultType calculate_result();
    void modify_somehow(const Param& p);

    // ...
```

```
  public:
    DoSomethingComplexAndOrExpensive(Param p) : state_changed(true) {
        modify_somehow(p);
    }

    void change_state(Param p) {
        modify_somehow(p);
        state_changed = true;
    }

    // Return some complex and/or expensive-to-calculate result.
    // As this has no reason to modify logical state, it is marked as "const".
    ResultType get_result() const;
};
ResultType DoSomethingComplexAndOrExpensive::get_result() const {
    // cached_result and state_changed can be modified, even with a const "this" pointer.
    // Even though the function doesn't modify logical state, it does modify physical state
    //  by caching the result, so it doesn't need to be recalculated every time the function
    //  is called.  This is indicated by cached_result and state_changed being mutable.

    if (state_changed) {
        cached_result = calculate_result();
        state_changed = false;
    }

    return cached_result;
}
```

Beachten Sie, dass Sie technisch nutzen *könnten* `const_cast` auf `this` , um es nicht-cv-qualifiziert zu machen, sollten Sie wirklich, ***wirklich*** nicht, und sollte verwenden `mutable` statt. Ein `const_cast` haftet nicht definiertes Verhalten aufzurufen , wenn auf ein Objekt verwendet , die tatsächlich *ist* `const` , während `mutable` ausgelegt ist , um sicher zu sein zu verwenden. Es ist jedoch möglich, dass Sie in sehr altem Code darauf stoßen.

Eine Ausnahme von dieser Regel ist das Definieren nicht-qualifizierter Zugriffsmethoden in Bezug auf `const` Zugriffsmethoden. Da das Objekt garantiert nicht `const` wenn die nicht für den CV geeignete Version aufgerufen wird, besteht kein UB-Risiko.

```
class CVAccessor {
    int arr[5];

  public:
    const int& get_arr_element(size_t i) const { return arr[i]; }

    int& get_arr_element(size_t i) {
        return const_cast<int&>(const_cast<const CVAccessor*>(this)->get_arr_element(i));
    }
};
```

Dies verhindert unnötige Duplizierungen von Code.

Wenn `this volatile` (einschließlich `const volatile`), wird es wie bei normalen Zeigern bei jedem Zugriff aus dem Speicher geladen, anstatt zwischengespeichert zu werden. Dies hat die gleichen Auswirkungen auf die Optimierung wie die Angabe eines anderen Zeigers als `volatile` , weshalb Vorsicht geboten ist.

Beachten Sie, dass , wenn eine Instanz cv-qualifiziert ist, ist es die einzige Mitglied Funktionen zugreifen , sind Funktionen , dessen Mitglieds ist erlaubt `this` Zeiger ist mindestens so cv-qualifiziert als Instanz selbst:

- Nicht-CV-Instanzen können auf alle Member-Funktionen zugreifen.
- `const` Instanzen können auf `const volatile` Funktionen von `const` und `const volatile` zugreifen.
- `volatile` Instanzen können auf `volatile` und `const volatile` Funktionen zugreifen.
- `const volatile` Instanzen können auf `const volatile` Funktionen zugreifen.

Dies ist einer der wichtigsten Grundsätze der `const` Korrektheit .

```
struct CVAccess {
    void    func()                {}
    void  func_c() const          {}
    void  func_v() volatile       {}
    void func_cv() const volatile {}
};

CVAccess cva;
cva.func();    // Good.
cva.func_c();  // Good.
cva.func_v();  // Good.
cva.func_cv(); // Good.

const CVAccess c_cva;
c_cva.func();    // Error.
c_cva.func_c();  // Good.
c_cva.func_v();  // Error.
c_cva.func_cv(); // Good.

volatile CVAccess v_cva;
v_cva.func();    // Error.
v_cva.func_c();  // Error.
v_cva.func_v();  // Good.
v_cva.func_cv(); // Good.

const volatile CVAccess cv_cva;
cv_cva.func();    // Error.
cv_cva.func_c();  // Error.
cv_cva.func_v();  // Error.
cv_cva.func_cv(); // Good.
```

diese Pointer Ref-Qualifiers

C ++ 11

Ähnlich wie bei `this` CV-Qualifiers können wir auch *Ref-Qualifier* auf `*this` anwenden. Ref-Qualifizierer verwendet , um zwischen normal und rvalue Referenz Semantik zu wählen, so dass der Compiler entweder Kopie verwenden oder Semantik je nachdem , welche besser geeignet bewegen, und werden angewandt `*this` anstelle von `this` .

Beachten Sie, dass trotz Ref-Qualifiers, die die Referenzsyntax verwenden, `this` selbst immer noch ein Zeiger ist. Beachten Sie auch, dass ref-Qualifier den Typ von `*this` nicht wirklich ändern.

Es ist nur einfacher, ihre Auswirkungen zu beschreiben und zu verstehen, indem man sie so betrachtet, als ob sie es taten.

```
struct RefQualifiers {
    std::string s;

    RefQualifiers(const std::string& ss = "The nameless one.") : s(ss) {}

    // Normal version.
    void func() &  { std::cout << "Accessed on normal instance "    << s << std::endl; }
    // Rvalue version.
    void func() && { std::cout << "Accessed on temporary instance " << s << std::endl; }

    const std::string& still_a_pointer() &  { return this->s; }
    const std::string& still_a_pointer() && { this->s = "Bob"; return this->s; }
};

// ...

RefQualifiers rf("Fred");
rf.func();              // Output:  Accessed on normal instance Fred
RefQualifiers{}.func(); // Output:  Accessed on temporary instance The nameless one
```

Eine Member-Funktion kann weder mit als auch ohne Ref-Qualifier Überladungen haben. Der Programmierer muss zwischen dem einen oder dem anderen wählen. Zum Glück können cv-qualifiers in Verbindung mit ref-qualifiers verwendet werden, so dass die `const` Richtigkeitsregeln befolgt werden können.

```
struct RefCV {
    void func() &                {}
    void func() &&               {}
    void func() const&           {}
    void func() const&&          {}
    void func() volatile&        {}
    void func() volatile&&       {}
    void func() const volatile&  {}
    void func() const volatile&& {}
};
```

Der This Pointer online lesen: https://riptutorial.com/de/cplusplus/topic/7146/der-this-pointer

Kapitel 37: Designmuster-Implementierung in C ++

Einführung

Auf dieser Seite finden Sie Beispiele, wie Entwurfsmuster in C ++ implementiert werden. Einzelheiten zu diesen Mustern finden Sie in der Dokumentation zu den Entwurfsmustern .

Bemerkungen

Ein Entwurfsmuster ist eine allgemein wiederverwendbare Lösung für ein häufig auftretendes Problem in einem bestimmten Kontext des Softwareentwurfs.

Examples

Beobachtermuster

Observer Pattern hat die Absicht, eine Eins-zu-Viele-Abhängigkeit zwischen Objekten zu definieren, sodass alle abhängigen Objekte benachrichtigt werden und automatisch aktualisiert werden, wenn sich ein Objekt ändert.

Das Subjekt und die Beobachter definieren die Eins-zu-Viele-Beziehung. Die Beobachter sind vom Motiv abhängig, so dass die Beobachter benachrichtigt werden, wenn sich der Zustand des Probanden ändert. Je nach Benachrichtigung können die Beobachter auch mit neuen Werten aktualisiert werden.

Hier ist das Beispiel aus dem Buch "Design Patterns" von Gamma.

```
#include <iostream>
#include <vector>

class Subject;

class Observer
{
public:
    virtual ~Observer() = default;
    virtual void Update(Subject&) = 0;
};

class Subject
{
public:
     virtual ~Subject() = default;
     void Attach(Observer& o) { observers.push_back(&o); }
     void Detach(Observer& o)
     {
         observers.erase(std::remove(observers.begin(), observers.end(), &o));
```

```
    }
    void Notify()
    {
        for (auto* o : observers) {
            o->Update(*this);
        }
    }
private:
    std::vector<Observer*> observers;
};

class ClockTimer : public Subject
{
public:

    void SetTime(int hour, int minute, int second)
    {
        this->hour = hour;
        this->minute = minute;
        this->second = second;

        Notify();
    }

    int GetHour() const { return hour; }
    int GetMinute() const { return minute; }
    int GetSecond() const { return second; }

private:
    int hour;
    int minute;
    int second;
};

class DigitalClock: public Observer
{
public:
    explicit DigitalClock(ClockTimer& s) : subject(s) { subject.Attach(*this); }
    ~DigitalClock() { subject.Detach(*this); }
    void Update(Subject& theChangedSubject) override
    {
        if (&theChangedSubject == &subject) {
            Draw();
        }
    }

    void Draw()
    {
        int hour = subject.GetHour();
        int minute = subject.GetMinute();
        int second = subject.GetSecond();

        std::cout << "Digital time is " << hour << ":"
                  << minute << ":"
                  << second << std::endl;
    }

private:
    ClockTimer& subject;
};
```

```
class AnalogClock: public Observer
{
public:
     explicit AnalogClock(ClockTimer& s) : subject(s) { subject.Attach(*this); }
     ~AnalogClock() { subject.Detach(*this); }
     void Update(Subject& theChangedSubject) override
     {
         if (&theChangedSubject == &subject) {
             Draw();
         }
     }
     void Draw()
     {
         int hour = subject.GetHour();
         int minute = subject.GetMinute();
         int second = subject.GetSecond();

         std::cout << "Analog time is " << hour << ":"
                   << minute << ":"
                   << second << std::endl;
     }
private:
     ClockTimer& subject;
};

int main()
{
    ClockTimer timer;

    DigitalClock digitalClock(timer);
    AnalogClock analogClock(timer);

    timer.SetTime(14, 41, 36);
}
```

Ausgabe:

```
Digital time is 14:41:36
Analog time is 14:41:36
```

Hier ist die Zusammenfassung des Musters:

1. Objekte (`DigitalClock` oder `AnalogClock` Objekt) verwenden , um den Betreff - Schnittstellen (`Attach()` oder `Detach()`) entweder abonnieren (Register) als Beobachter oder abmelden (entfernen) sich davon , dass Beobachter (`subject.Attach(*this);` , `subject.Detach(*this);`

2. Jedes Subjekt kann viele Beobachter haben (`vector<Observer*> observers;`).

3. Alle Beobachter müssen die Observer-Schnittstelle implementieren. Diese Schnittstelle hat nur eine Methode, `Update()` , die aufgerufen wird, wenn sich der Status des Betreffs ändert (`Update(Subject &)`).

4. Zusätzlich zu den Methoden `Attach()` und `Detach()` implementiert das konkrete Subjekt eine `Notify()` Methode, mit der alle aktuellen Beobachter aktualisiert werden, wenn sich der Status ändert. In diesem Fall werden jedoch alle in der übergeordneten Klasse `Subject` (`Subject::Attach (Observer&)` , `void Subject::Detach(Observer&)` und `void Subject::Notify()` .

5. Das Concrete-Objekt kann auch über Methoden zum Setzen und Abrufen seines Status verfügen.

6. Konkrete Beobachter können jede Klasse sein, die die Observer-Schnittstelle implementiert. Jeder Beobachter abonniert (registrieren) sich mit einem konkreten Thema, um ein Update zu erhalten (`subject.Attach(*this);`).

7. Die beiden Objekte des Observer Patterns sind **lose miteinander verbunden** , sie können miteinander interagieren, jedoch nur wenig voneinander wissen.

Variation:

Signal und Slots

Signale und Slots ist ein in Qt eingeführtes Sprachkonstrukt, das die Implementierung des Observer-Musters vereinfacht und den Boilerplate-Code vermeidet. Das Konzept besteht darin, dass Steuerelemente (auch als Widgets bezeichnet) Signale mit Ereignisinformationen senden können, die von anderen Steuerelementen mithilfe spezieller Funktionen (Slots) empfangen werden können. Der Slot in Qt muss ein als solches deklariertes Klassenmitglied sein. Das Signal / Slot-System passt gut zu der Art und Weise, wie grafische Benutzeroberflächen entworfen werden. In ähnlicher Weise kann das Signal / Slot-System für asynchrone E / A (einschließlich Sockets, Pipes, serielle Geräte usw.) Ereignisbenachrichtigung verwendet werden, oder um Zeitüberschreitungsereignisse mit geeigneten Objektinstanzen und Methoden oder Funktionen zu verknüpfen. Es muss kein Registrierungs- / Deregistrierungs- / Aufrufcode geschrieben werden, da der Meta Object Compiler (MOC) von Qt automatisch die benötigte Infrastruktur generiert.

Die C # -Sprache unterstützt auch ein ähnliches Konstrukt, jedoch mit einer anderen Terminologie und Syntax: Ereignisse spielen die Rolle von Signalen und Delegaten sind die Slots. Darüber hinaus kann ein Delegat eine lokale Variable sein, ähnlich wie ein Funktionszeiger, während ein Slot in Qt ein als solcher deklarierter Klassenmitglied sein muss.

Adaptermuster

Konvertieren Sie das Interface einer Klasse in ein anderes Interface, das die Clients erwarten. Mit Adapter (oder Wrapper) können Klassen zusammenarbeiten, die aufgrund von inkompatiblen Schnittstellen sonst nicht möglich wären. Die Motivation des Adapter-Patterns ist, dass wir vorhandene Software wiederverwenden können, wenn wir die Schnittstelle ändern können.

1. Das Adaptermuster hängt von der Objektzusammensetzung ab.

2. Client ruft den Vorgang für das Adapterobjekt auf.

3. Der Adapter ruft Adaptee auf, um die Operation auszuführen.

4. In STL, Stack vom Vektor angepasst: Wenn Stack push () ausführt, macht der darunterliegende Vektor vector :: push_back ().

Beispiel:

```
#include <iostream>

// Desired interface (Target)
class Rectangle
{
  public:
    virtual void draw() = 0;
};

// Legacy component (Adaptee)
class LegacyRectangle
{
  public:
    LegacyRectangle(int x1, int y1, int x2, int y2) {
        x1_ = x1;
        y1_ = y1;
        x2_ = x2;
        y2_ = y2;
        std::cout << "LegacyRectangle(x1,y1,x2,y2)\n";
    }
    void oldDraw() {
        std::cout << "LegacyRectangle:  oldDraw(). \n";
    }
  private:
    int x1_;
    int y1_;
    int x2_;
    int y2_;
};

// Adapter wrapper
class RectangleAdapter: public Rectangle, private LegacyRectangle
{
  public:
    RectangleAdapter(int x, int y, int w, int h):
      LegacyRectangle(x, y, x + w, y + h) {
         std::cout << "RectangleAdapter(x,y,x+w,x+h)\n";
      }

    void draw() {
        std::cout << "RectangleAdapter: draw().\n";
        oldDraw();
    }
};

int main()
{
  int x = 20, y = 50, w = 300, h = 200;
  Rectangle *r = new RectangleAdapter(x,y,w,h);
  r->draw();
}

//Output:
//LegacyRectangle(x1,y1,x2,y2)
//RectangleAdapter(x,y,x+w,x+h)
```

Zusammenfassung des Codes:

1. Der Kunde meint, er rede mit einem `Rectangle`

2. Das Ziel ist die `Rectangle` Klasse. Daraufhin ruft der Client die Methode auf.

```
Rectangle *r = new RectangleAdapter(x,y,w,h);
r->draw();
```

3. Beachten Sie, dass die Adapterklasse mehrere Vererbung verwendet.

```
class RectangleAdapter: public Rectangle, private LegacyRectangle {
    ...
}
```

4. Mit dem Adapter `RectangleAdapter` das `LegacyRectangle` auf request (`draw()` auf einem `Rectangle`) `LegacyRectangle` , indem es BEIDE Klassen erbt.

5. Die `LegacyRectangle` Klasse verfügt nicht über die gleichen Methoden (`draw()`) wie `Rectangle` , der `Adapter(RectangleAdapter)` kann jedoch die Aufrufe der `Rectangle` Methode `LegacyRectangle` und die `oldDraw()` Methode `oldDraw()` .

```
class RectangleAdapter: public Rectangle, private LegacyRectangle {
  public:
    RectangleAdapter(int x, int y, int w, int h):
      LegacyRectangle(x, y, x + w, y + h) {
        std::cout << "RectangleAdapter(x,y,x+w,x+h)\n";
      }

    void draw() {
        std::cout << "RectangleAdapter: draw().\n";
        oldDraw();
    }
};
```

Das Design des **Adapters** übersetzt die Schnittstelle für eine Klasse in eine kompatible, aber andere Schnittstelle. Dies ähnelt dem **Proxy-** Muster, da es sich um einen Einkomponenten-Wrapper handelt. Die Schnittstelle für die Adapterklasse und die ursprüngliche Klasse kann jedoch unterschiedlich sein.

Wie im obigen Beispiel gezeigt wurde, ist dieses **Adaptermuster** hilfreich, um eine andere Schnittstelle für eine vorhandene API bereitzustellen, damit diese mit anderem Code arbeiten kann. Durch die Verwendung eines Adaptermusters können wir auch heterogene Schnittstellen verwenden und diese transformieren, um eine konsistente API bereitzustellen.

Bridge Pattern hat eine ähnliche Struktur wie ein Objektadapter, Bridge hat jedoch eine andere Absicht: Es soll eine Schnittstelle von ihrer Implementierung **trennen** , sodass sie leicht und unabhängig voneinander variiert werden kann. Ein **Adapter** soll **die Schnittstelle** eines **vorhandenen** Objekts **ändern** .

Fabrikmuster

Factory Pattern entkoppelt die Objekterstellung und ermöglicht die Erstellung anhand des Namens über eine gemeinsame Schnittstelle:

```
class Animal{
public:
    virtual std::shared_ptr<Animal> clone() const = 0;
    virtual std::string  getname() const = 0;
};

class Bear: public Animal{
public:
    virtual std::shared_ptr<Animal> clone() const override
    {
        return std::make_shared<Bear>(*this);
    }
    virtual std::string getname() const override
    {
        return "bear";
    }
};


class Cat: public Animal{
public:
    virtual std::shared_ptr<Animal> clone() const override
    {
        return std::make_shared<Cat>(*this);
    }
    virtual std::string  getname() const override
    {
        return "cat";
    }
};

class AnimalFactory{
public:
    static std::shared_ptr<Animal> getAnimal( const std::string&   name )
    {
      if ( name == "bear" )
        return std::make_shared<Bear>();
      if ( name == "cat" )
        return std::shared_ptr<Cat>();

    return nullptr;
    }


};
```

Builder Pattern mit fließender API

Das Builder Pattern koppelt die Erstellung des Objekts vom Objekt selbst ab. Die Grundidee dahinter ist, dass **ein Objekt nicht für seine eigene Erstellung verantwortlich sein muss** . Die korrekte und gültige Assembly eines komplexen Objekts kann an sich eine komplizierte Aufgabe sein, daher kann diese Aufgabe an eine andere Klasse delegiert werden.

Inspiriert durch den Email Builder in C # habe ich mich entschlossen, hier eine C ++ - Version zu erstellen. Ein E-Mail-Objekt ist nicht unbedingt ein *sehr komplexes Objekt* , es kann jedoch das Muster demonstrieren.

```
#include <iostream>
#include <sstream>
#include <string>

using namespace std;

// Forward declaring the builder
class EmailBuilder;

class Email
{
  public:
    friend class EmailBuilder;  // the builder can access Email's privates

    static EmailBuilder make();

    string to_string() const {
        stringstream stream;
        stream << "from: " << m_from
               << "\nto: " << m_to
               << "\nsubject: " << m_subject
               << "\nbody: " << m_body;
        return stream.str();
    }

  private:
    Email() = default; // restrict construction to builder

    string m_from;
    string m_to;
    string m_subject;
    string m_body;
};

class EmailBuilder
{
  public:
    EmailBuilder& from(const string &from) {
        m_email.m_from = from;
        return *this;
    }

    EmailBuilder& to(const string &to) {
        m_email.m_to = to;
        return *this;
    }

    EmailBuilder& subject(const string &subject) {
        m_email.m_subject = subject;
        return *this;
    }

    EmailBuilder& body(const string &body) {
        m_email.m_body = body;
        return *this;
    }

    operator Email&&() {
        return std::move(m_email); // notice the move
    }
```

```
  private:
    Email m_email;
};

EmailBuilder Email::make()
{
    return EmailBuilder();
}

// Bonus example!
std::ostream& operator <<(std::ostream& stream, const Email& email)
{
    stream << email.to_string();
    return stream;
}


int main()
{
    Email mail = Email::make().from("me@mail.com")
                              .to("you@mail.com")
                              .subject("C++ builders")
                              .body("I like this API, don't you?");

    cout << mail << endl;
}
```

Bei älteren Versionen von C ++ kann man den `std::move` Vorgang einfach ignorieren und das && aus dem Konvertierungsoperator entfernen (obwohl dies eine temporäre Kopie erstellt).

Der Builder beendet seine Arbeit, wenn er die erstellte E-Mail vom `operator Email&&()` freigibt. In diesem Beispiel ist der Builder ein temporäres Objekt und gibt die E-Mail zurück, bevor sie gelöscht wird. Sie können auch eine explizite Operation wie `Email EmailBuilder::build() {...}` anstelle des Konvertierungsoperators verwenden.

Führe den Erbauer herum

Ein großartiges Feature des Builder Patterns ist die Möglichkeit, **mehrere Darsteller zu verwenden, um ein Objekt zusammen zu erstellen.** Dies geschieht, indem der Builder an die anderen Akteure übergeben wird, von denen jeder weitere Informationen zum erstellten Objekt liefert. Dies ist besonders nützlich, wenn Sie eine Art Abfrage erstellen und Filter und andere Spezifikationen hinzufügen.

```
void add_addresses(EmailBuilder& builder)
{
    builder.from("me@mail.com")
           .to("you@mail.com");
}

void compose_mail(EmailBuilder& builder)
{
    builder.subject("I know the subject")
           .body("And the body. Someone else knows the addresses.");
}
```

```
int main()
{
    EmailBuilder builder;
    add_addresses(builder);
    compose_mail(builder);

    Email mail = builder;
    cout << mail << endl;
}
```

Designvariante: Objekt veränderlich

Sie können das Design dieses Musters an Ihre Bedürfnisse anpassen. Ich werde eine Variante geben.

Im angegebenen Beispiel ist das Email-Objekt unveränderlich, dh seine Eigenschaften können nicht geändert werden, da auf sie nicht zugegriffen werden kann. Dies war eine gewünschte Funktion. Wenn Sie das Objekt nach seiner Erstellung ändern müssen, müssen Sie einige Setzer angeben. Da diese Setter im Builder dupliziert werden, können Sie dies in einer Klasse erledigen (es ist keine Builder-Klasse mehr erforderlich). Trotzdem würde ich die Notwendigkeit in Betracht ziehen, das gebaute Objekt überhaupt veränderbar zu machen.

Designmuster-Implementierung in C ++ online lesen:
https://riptutorial.com/de/cplusplus/topic/4335/designmuster-implementierung-in-c-plusplus

Kapitel 38: Die Regel von Drei, Fünf und Null

Examples

Fünfter Regel

C ++ 11

C ++ 11 führt zwei neue spezielle Memberfunktionen ein: den Bewegungskonstruktor und den Bewegungszuweisungsoperator. Aus den gleichen Gründen, die Sie der Drei-Regel in C ++ 03 folgen möchten, möchten Sie normalerweise der Fünf-Regel in C ++ 11 folgen: Wenn eine Klasse EINE von fünf speziellen Member-Funktionen erfordert, und verschieben Sie die Semantik erwünscht sind, dann werden höchstwahrscheinlich ALLE FÜNF von ihnen benötigt.

Beachten Sie jedoch, dass das Nichtbefolgen der Fünf-Regel normalerweise nicht als Fehler, sondern als verpasste Optimierungsmöglichkeit betrachtet wird, solange die Drei-Regel weiterhin befolgt wird. Wenn kein Verschiebungskonstruktor oder Verschiebungszuweisungsoperator verfügbar ist, wenn der Compiler normalerweise einen verwenden würde, verwendet er stattdessen die Kopiersemantik, was zu einer weniger effizienten Operation aufgrund unnötiger Kopiervorgänge führt. Wenn für eine Klasse keine Verschiebesemantik gewünscht wird, muss kein Verschiebungskonstruktor oder Zuweisungsoperator deklariert werden.

Gleiches Beispiel wie für die Dreierregel:

```
class Person
{
    char* name;
    int age;

public:
    // Destructor
    ~Person() { delete [] name; }

    // Implement Copy Semantics
    Person(Person const& other)
        : name(new char[std::strlen(other.name) + 1])
        , age(other.age)
    {
        std::strcpy(name, other.name);
    }

    Person &operator=(Person const& other)
    {
        // Use copy and swap idiom to implement assignment.
        Person copy(other);
        swap(*this, copy);
        return *this;
    }

    // Implement Move Semantics
    // Note: It is usually best to mark move operators as noexcept
    //       This allows certain optimizations in the standard library
```

```
    //       when the class is used in a container.

    Person(Person&& that) noexcept
        : name(nullptr)               // Set the state so we know it is undefined
        , age(0)
    {
        swap(*this, that);
    }

    Person& operator=(Person&& that) noexcept
    {
        swap(*this, that);
        return *this;
    }

    friend void swap(Person& lhs, Person& rhs) noexcept
    {
        std::swap(lhs.name, rhs.name);
        std::swap(lhs.age, rhs.age);
    }
};
```

Alternativ können sowohl der Kopier- als auch der Verschiebungszuweisungsoperator durch einen einzelnen Zuweisungsoperator ersetzt werden, der anstelle des Verweises oder des R-Werts einen Wert als Instanz verwendet, um die Verwendung des Copy-and-Swap-Idioms zu erleichtern.

```
Person& operator=(Person copy)
{
    swap(*this, copy);
    return *this;
}
```

Die Erweiterung von der Drei-Regel auf die Fünf-Regel ist aus Leistungsgründen wichtig, aber in den meisten Fällen nicht unbedingt erforderlich. Durch das Hinzufügen des Kopierkonstruktors und des Zuweisungsoperators wird sichergestellt, dass durch das Verschieben des Typs kein Speicherplatzverlust entsteht (in diesem Fall greift Move-Construing einfach auf das Kopieren zurück), führt jedoch Kopien aus, die der Aufrufer wahrscheinlich nicht erwartet hat.

Nullregel

C ++ 11

Wir können die Prinzipien der Regel von Fünf und von RAII kombinieren, um eine viel schlankere Schnittstelle zu erhalten: Die Regel von Null: Jede Ressource, die verwaltet werden muss, sollte in einem eigenen Typ sein. Dieser Typ muss der Fünf-Regel folgen, aber alle Benutzer dieser Ressource müssen *keine* der fünf speziellen Member-Funktionen schreiben und können einfach alle als `default` festlegen.

Mit der im Drei-Regel-Beispiel eingeführten `Person` Klasse können wir ein Ressourcenverwaltungsobjekt für `cstrings` :

```
class cstring {
private:
```

```
    char* p;

public:
    ~cstring() { delete [] p; }
    cstring(cstring const& );
    cstring(cstring&& );
    cstring& operator=(cstring const& );
    cstring& operator=(cstring&& );

    /* other members as appropriate */
};
```

Und wenn dies einmal getrennt ist, wird unsere `Person` Klasse viel einfacher:

```
class Person {
    cstring name;
    int arg;

public:
    ~Person() = default;
    Person(Person const& ) = default;
    Person(Person&& ) = default;
    Person& operator=(Person const& ) = default;
    Person& operator=(Person&& ) = default;

    /* other members as appropriate */
};
```

Die besonderen Mitglieder in `Person` müssen nicht einmal ausdrücklich deklariert werden; Der Compiler setzt sie entsprechend dem Inhalt von `Person` standardmäßig zurück oder löscht sie entsprechend. Daher ist das Folgende auch ein Beispiel für die Nullregel.

```
struct Person {
    cstring name;
    int arg;
};
```

Wenn es sich bei `cstring` um einen Nur-Move-Typ mit einem Konstruktor / Zuweisungsoperator zum `delete` Kopien handeln würde, würde `Person` automatisch ebenfalls nur Move sein.

Die Term-Null-Regel wurde von R. Martinho Fernandes eingeführt

Dreierregel

c ++ 03

Die Dreierregel besagt, dass, wenn ein Typ jemals über einen benutzerdefinierten Kopierkonstruktor, einen Kopierzuweisungsoperator oder einen Destruktor verfügen muss, *alle drei vorhanden sein müssen* .

Der Grund für die Regel ist, dass eine Klasse, die eine der drei Klassen benötigt, eine Ressource (Dateizugriffsnummern, dynamisch zugewiesenen Speicher usw.) verwaltet, und alle drei werden benötigt, um diese Ressource konsistent zu verwalten. Die Kopierfunktionen behandeln, wie die

Ressource zwischen Objekten kopiert wird, und der Destruktor würde die Ressource gemäß den RAII-Prinzipien zerstören.

Betrachten Sie einen Typ, der eine Zeichenfolgenressource verwaltet:

```
class Person
{
    char* name;
    int age;

public:
    Person(char const* new_name, int new_age)
        : name(new char[std::strlen(new_name) + 1])
        , age(new_age)
    {
       std::strcpy(name, new_name);
    }

    ~Person() {
        delete [] name;
    }
};
```

Da der `name` im Konstruktor vergeben wurde, wird der Destruktor freigegeben, um Speicherlecks zu vermeiden. Was passiert aber, wenn ein solches Objekt kopiert wird?

```
int main()
{
    Person p1("foo", 11);
    Person p2 = p1;
}
```

Zuerst wird `p1` konstruiert. Dann wird `p2` von `p1` kopiert. Der von C ++ generierte Kopierkonstruktor kopiert jedoch jede Komponente des Typs unverändert. Was bedeutet, dass `p1.name` und `p2.name` auf **dieselbe** Zeichenfolge verweisen.

Wenn `main` endet, werden Destruktoren aufgerufen. Der erste Destruktor von `p2` wird aufgerufen. Die Zeichenfolge wird gelöscht. Dann wird der Destruktor von `p1` aufgerufen. Die Zeichenfolge ist jedoch *bereits gelöscht* . Das Aufrufen von `delete` in bereits gelöschtem Speicher führt zu undefiniertem Verhalten.

Um dies zu vermeiden, muss ein geeigneter Copy-Konstruktor bereitgestellt werden. Ein Ansatz besteht darin, ein Referenzzählsystem zu implementieren, bei dem verschiedene `Person` dieselben Zeichenfolgendaten verwenden. Bei jeder Ausführung einer Kopie wird der gemeinsame Referenzzähler erhöht. Der Destruktor verringert dann die Referenzzählung und gibt den Speicher nur dann frei, wenn die Zählung Null ist.

Oder wir könnten Wertsemantik und tiefes Kopierverhalten implementieren:

```
Person(Person const& other)
    : name(new char[std::strlen(other.name) + 1])
    , age(other.age)
{
```

```
    std::strcpy(name, other.name);
}

Person &operator=(Person const& other)
{
    // Use copy and swap idiom to implement assignment
    Person copy(other);
    swap(copy);            //  assume swap() exchanges contents of *this and copy
    return *this;
}
```

Die Implementierung des Kopierzuweisungsoperators wird durch die Notwendigkeit der Freigabe eines vorhandenen Puffers erschwert. Die Copy- und Swap-Technik erstellt ein temporäres Objekt, das einen neuen Puffer enthält. Durch das Austauschen des Inhalts von `*this` und `copy` das Eigentum an der `copy` des ursprünglichen Puffers erhalten. Bei der Zerstörung einer `copy` wird der Puffer, der zuvor von `*this` besessen wurde, `*this` .

Selbstzuteilung Schutz

Beim Schreiben eines Kopierzuweisungsoperators ist es *sehr* wichtig, dass er bei Selbstzuweisung arbeiten kann. Das heißt, es muss dies zulassen:

```
SomeType t = ...;
t = t;
```

Selbstzuteilung geschieht normalerweise nicht so offensichtlich. Es geschieht in der Regel über einen Umweg durch verschiedene Kontrollsysteme, in denen die Lage der Zuordnung hat einfach zwei `Person` Zeiger oder Verweise und hat keine Ahnung , dass sie das gleiche Objekt sind.

Jeder Kopierzuweisungsoperator, den Sie schreiben, muss dies berücksichtigen können.

Die typische Methode ist, die Zuweisungslogik in eine Bedingung wie folgt zu hüllen:

```
SomeType &operator=(const SomeType &other)
{
    if(this != &other)
    {
        //Do assignment logic.
    }
    return *this;
}
```

Hinweis: Es ist wichtig, über die Selbstzuweisung nachzudenken und sicherzustellen, dass sich Ihr Code bei Auftreten korrekt verhält. Selbstzuweisung ist jedoch ein sehr seltenes Ereignis und die Optimierung, um zu verhindern, dass dies den Normalfall pessimiert. Da der Normalfall sehr viel häufiger vorkommt, kann die Verbesserung der Code-Effizienz durch die Pessimierung der Selbstzuweisung sehr beeinträchtigt werden.

Die übliche Technik zum Implementieren des Zuweisungsoperators ist beispielsweise das `copy and swap idiom` . Die normale Implementierung dieser Technik macht sich nicht die Mühe, die Selbstzuweisung zu testen (obwohl Selbstzuweisung teuer ist, weil eine Kopie erstellt wird). Der

Grund ist, dass die Pessimisierung des Normalfalls viel kostspieliger ist (da dies öfter geschieht).

c ++ 11

Verschiebungszuweisungsoperatoren müssen auch vor Selbstzuweisung geschützt werden. Die Logik für viele dieser Operatoren basiert jedoch auf `std::swap` , das das Auslagern aus / in den gleichen Speicher problemlos handhaben kann. Wenn Ihre Zugzuweisungslogik nichts anderes als eine Reihe von Swap-Operationen ist, brauchen Sie keinen Schutz vor Selbstzuweisung.

Ist dies nicht der Fall, *müssen* Sie ähnliche Maßnahmen wie oben einleiten.

Die Regel von Drei, Fünf und Null online lesen: https://riptutorial.com/de/cplusplus/topic/1206/die-regel-von-drei--funf-und-null

Kapitel 39: Eigenschaften eingeben

Bemerkungen

Typmerkmale sind Templatkonstrukte, mit denen die Eigenschaften verschiedener Typen zur Kompilierzeit verglichen und getestet werden. Sie können verwendet werden, um eine bedingte Logik zur Kompilierzeit bereitzustellen, die die Funktionalität Ihres Codes auf bestimmte Weise einschränken oder erweitern kann. Die Typmerkmalsbibliothek wurde mit dem Standard `c++11` der eine Reihe verschiedener Funktionalitäten bietet. Es ist auch möglich, eigene Vorlagen für den Vergleich von Typenmerkmalen zu erstellen.

Examples

Standardmerkmale

C ++ 11

Der `type_traits` Header enthält eine Reihe von Vorlagenklassen und Helfern, mit denen die Eigenschaften von Typen `type_traits` der Kompilierung transformiert und überprüft werden können.

Diese Merkmale werden normalerweise in Vorlagen verwendet, um auf Benutzerfehler zu prüfen, generische Programmierung zu unterstützen und Optimierungen zu ermöglichen.

Die meisten Typmerkmale werden verwendet, um zu überprüfen, ob ein Typ einige Kriterien erfüllt. Diese haben folgende Form:

```
template <class T> struct is_foo;
```

Wenn die Vorlagenklasse mit einem Typ instanziiert wird, der einige Kriterien `foo` erfüllt, erbt `is_foo<T>` von `std::integral_constant<bool,true>` (aka `std::true_type`), andernfalls erbt sie von `std::integral_constant<bool,false>` (auch bekannt als `std::false_type`). Dies gibt dem Merkmal die folgenden Mitglieder:

Konstanten

`static constexpr bool value`

`true` wenn `T` die Kriterien `foo` erfüllt, andernfalls `false`

Funktionen

`operator bool`

Gibt den `value`

C ++ 14

`bool operator()`

Gibt den `value`

Typen

Name	Definition
value_type	bool
type	std::integral_constant<bool,value>

Das Merkmal kann dann in Konstrukten wie `static_assert` oder `std::enable_if` . Ein Beispiel mit `std::is_pointer` :

```
template <typename T>
void i_require_a_pointer (T t) {
    static_assert(std::is_pointer<T>::value, "T must be a pointer type");
}

//Overload for when T is not a pointer type
template <typename T>
typename std::enable_if<!std::is_pointer<T>::value>::type
does_something_special_with_pointer (T t) {
    //Do something boring
}

//Overload for when T is a pointer type
template <typename T>
typename std::enable_if<std::is_pointer<T>::value>::type
does_something_special_with_pointer (T t) {
    //Do something special
}
```

Es gibt auch verschiedene Merkmale, die Typen transformieren, z. B. `std::add_pointer` und `std::underlying_type` . Diese Merkmale stellen im Allgemeinen einen einzelnen `type` bereit, der den transformierten Typ enthält. Beispiel: `std::add_pointer<int>::type` ist `int*` .

Geben Sie Relations mit std :: is_same ein

C ++ 11

Die Beziehung `std::is_same<T, T>` wird zum Vergleich zweier Typen verwendet. Es wird als boolean ausgewertet, true, wenn die Typen gleich sind, andernfalls false.

z.B

```
// Prints true on most x86 and x86_64 compilers.
std::cout << std::is_same<int, int32_t>::value << "\n";
// Prints false on all compilers.
std::cout << std::is_same<float, int>::value << "\n";
// Prints false on all compilers.
std::cout  << std::is_same<unsigned int, int>::value << "\n";
```

Die `std::is_same` funktioniert auch unabhängig von Typedefs. Dies wird im ersten Beispiel beim Vergleich von `int == int32_t` demonstriert. Dies ist jedoch nicht ganz klar.

z.B

```
// Prints true on all compilers.
typedef int MyType
std::cout << std::is_same<int, MyType>::value <<  "\n";
```

Verwenden von `std::is_same` zum `std::is_same` , wenn eine Klasse oder Funktion mit Vorlagen nicht ordnungsgemäß verwendet wird.

In Kombination mit einer statischen `std::is_same` kann die Vorlage `std::is_same` ein wertvolles Werkzeug sein, um die ordnungsgemäße Verwendung von Klassen und Funktionen mit Vorlagen zu erzwingen.

ZB eine Funktion, die nur Eingaben von einem `int` und die Wahl von zwei Strukturen erlaubt.

```
#include <type_traits>
struct foo {
  int member;
  // Other variables
};

struct bar {
  char member;
};

template<typename T>
int AddStructMember(T var1, int var2) {
  // If type T != foo || T != bar then show error message.
  static_assert(std::is_same<T, foo>::value ||
    std::is_same<T, bar>::value,
    "This function does not support the specified type.");
  return var1.member + var2;
}
```

Grundtypeigenschaften

C ++ 11

Es gibt eine Reihe verschiedener Typmerkmale, die allgemeinere Typen vergleichen.

Ist Integral:

Wird für alle Integer-Typen `int` , `char` , `long` , `unsigned int` usw. als wahr `unsigned int` .

```
std::cout << std::is_integral<int>::value << "\n"; // Prints true.
std::cout << std::is_integral<char>::value << "\n"; // Prints true.
std::cout << std::is_integral<float>::value << "\n"; // Prints false.
```

Ist Floating Point:

Wird für alle Gleitkommatypen als wahr ausgewertet. `float` , `double` , `long double` usw.

```
std::cout << std::is_floating_point<float>::value << "\n"; // Prints true.
std::cout << std::is_floating_point<double>::value << "\n"; // Prints true.
std::cout << std::is_floating_point<char>::value << "\n"; // Prints false.
```

Ist Enum:

Wird für alle Aufzählungstypen, einschließlich der Aufzählungsklasse, als wahr `enum class` .

```
enum fruit {apple, pair, banana};
enum class vegetable {carrot, spinach, leek};
std::cout << std::is_enum<fruit>::value << "\n"; // Prints true.
std::cout << std::is_enum<vegetable>::value << "\n"; // Prints true.
std::cout << std::is_enum<int>::value << "\n"; // Prints false.
```

Ist Zeiger:

Bewertet für alle Zeiger als wahr.

```
std::cout << std::is_pointer<int *>::value << "\n"; // Prints true.
typedef int* MyPTR;
std::cout << std::is_pointer<MyPTR>::value << "\n"; // Prints true.
std::cout << std::is_pointer<int>::value << "\n"; // Prints false.
```

Ist Klasse:

Wird für alle Klassen und struct mit Ausnahme von `enum class` als wahr `enum class` .

```
struct FOO {int x, y;};
class BAR {
 public:
  int x, y;
};
enum class fruit {apple, pair, banana};
std::cout << std::is_class<FOO>::value << "\n"; // Prints true.
std::cout << std::is_class<BAR>::value << "\n"; // Prints true.
std::cout << std::is_class<fruit>::value << "\n"; // Prints false.
std::cout << std::is_class<int>::value << "\n"; // Prints false.
```

Geben Sie Eigenschaften ein

C ++ 11

Typeneigenschaften vergleichen die Modifizierer, die für verschiedene Variablen platziert werden können. Die Nützlichkeit dieser Merkmale ist nicht immer offensichtlich.

Hinweis: Das folgende Beispiel bietet nur eine Verbesserung eines nicht optimierenden Compilers. Es ist eher ein Beweis für das Konzept als ein komplexes Beispiel.

zB schnelle Division durch vier.

```
template<typename T>
inline T FastDivideByFour(cont T &var) {
  // Will give an error if the inputted type is not an unsigned integral type.
  static_assert(std::is_unsigned<T>::value && std::is_integral<T>::value,
    "This function is only designed for unsigned integral types.");
  return (var >> 2);
}
```

Ist konstant:

Dies wird als wahr ausgewertet, wenn type konstant ist.

```
std::cout << std::is_const<const int>::value << "\n"; // Prints true.
std::cout << std::is_const<int>::value << "\n"; // Prints false.
```

Ist flüchtig:

Dies wird als wahr ausgewertet, wenn der Typ flüchtig ist.

```
std::cout << std::is_volatile<static volatile int>::value << "\n"; // Prints true.
std::cout << std::is_const<const int>::value << "\n"; // Prints false.
```

Ist unterschrieben:

Dies wird für alle signierten Typen als wahr bewertet.

```
std::cout << std::is_signed<int>::value << "\n"; // Prints true.
std::cout << std::is_signed<float>::value << "\n"; // Prints true.
std::cout << std::is_signed<unsigned int>::value << "\n"; // Prints false.
std::cout << std::is_signed<uint8_t>::value << "\n"; // Prints false.
```

Ist unsigniert:

Wird für alle vorzeichenlosen Typen als wahr ausgewertet.

```
std::cout << std::is_unsigned<unsigned int>::value << "\n"; // Prints true.
std::cout << std::is_signed<uint8_t>::value << "\n"; // Prints true.
std::cout << std::is_unsigned<int>::value << "\n"; // Prints false.
std::cout << std::is_signed<float>::value << "\n"; // Prints false.
```

Eigenschaften eingeben online lesen:
https://riptutorial.com/de/cplusplus/topic/4750/eigenschaften-eingeben

Kapitel 40: Eine Definitionsregel (ODR)

Examples

Mehrfach definierte Funktion

Die wichtigste Konsequenz der One-Definition-Regel ist, dass Nicht-Inline-Funktionen mit externer Verknüpfung nur einmal in einem Programm definiert werden sollten, obwohl sie mehrmals deklariert werden können. Daher sollten solche Funktionen nicht in Headern definiert werden, da ein Header aus verschiedenen Übersetzungseinheiten mehrfach eingefügt werden kann.

`foo.h` :

```
#ifndef FOO_H
#define FOO_H
#include <iostream>
void foo() { std::cout << "foo"; }
void bar();
#endif
```

`foo.cpp` :

```
#include "foo.h"
void bar() { std:: cout << "bar"; }
```

`main.cpp` :

```
#include "foo.h"
int main() {
    foo();
    bar();
}
```

In diesem Programm ist die Funktion `foo` im Header `foo.h` , der zweimal enthalten ist: einmal aus `foo.cpp` und einmal aus `main.cpp` . Jede Übersetzungseinheit enthält daher ihre eigene Definition von `foo` . Beachten Sie, dass die Include-Guards in `foo.h` dies nicht verhindern, da `foo.cpp` und `main.cpp` beide `foo.h` *separat* enthalten. Das wahrscheinlichste Ergebnis des Versuchs, dieses Programm zu erstellen, ist ein Verbindungszeitfehler, der `foo` als mehrfach definiert identifiziert.

Um solche Fehler zu vermeiden, sollte man Funktionen in Header *deklariert* und *definiert* sie in den entsprechenden `.cpp` - Dateien, mit einigen Ausnahmen (andere siehe Beispiele).

Inline-Funktionen

Eine `inline` deklarierte Funktion kann in mehreren Übersetzungseinheiten definiert werden, vorausgesetzt, dass alle Definitionen identisch sind. Es muss auch in jeder Übersetzungseinheit definiert werden, in der es verwendet wird. Inline-Funktionen *sollten* daher in Kopfzeilen definiert

werden und müssen nicht in der Implementierungsdatei erwähnt werden.

Das Programm verhält sich so, als ob es eine einzige Definition der Funktion gibt.

`foo.h` :

```
#ifndef FOO_H
#define FOO_H
#include <iostream>
inline void foo() { std::cout << "foo"; }
void bar();
#endif
```

`foo.cpp` :

```
#include "foo.h"
void bar() {
    // more complicated definition
}
```

`main.cpp` :

```
#include "foo.h"
int main() {
    foo();
    bar();
}
```

In diesem Beispiel ist die einfachere Funktion `foo` wird inline in der Header - Datei definiert , während das kompliziertere Funktion `bar` nicht inline ist und in der Implementierungsdatei definiert. Sowohl die `foo.cpp` und `main.cpp` Übersetzungseinheiten enthalten Definitionen von `foo` , aber dieses Programm ist gut gebildet , da `foo` inline ist.

Eine in einer Klassendefinition definierte Funktion (die eine Member- oder eine Friend-Funktion sein kann) ist *implizit* inline. Wenn also eine Klasse in einem Header definiert ist, können Member-Funktionen der Klasse innerhalb der Klassendefinition definiert werden, auch wenn die Definitionen in mehreren Übersetzungseinheiten enthalten sind:

```
// in foo.h
class Foo {
    void bar() { std::cout << "bar"; }
    void baz();
};

// in foo.cpp
void Foo::baz() {
   // definition
}
```

Die Funktion `Foo::baz` ist als Out-of-Line definiert, es handelt sich also *nicht um* eine Inline-Funktion und darf nicht im Header definiert werden.

ODR-Verletzung durch Überlastlösung

Selbst bei identischen Token für Inline-Funktionen kann ODR verletzt werden, wenn das Nachschlagen von Namen nicht auf dieselbe Entität verweist. Betrachten wir `func` im Folgenden:

- header.h

```
void overloaded(int);
inline void func() { overloaded('*'); }
```

- foo.cpp

```
#include "header.h"

void foo()
{
    func(); // `overloaded` refers to `void overloaded(int)`
}
```

- bar.cpp

```
void overloaded(char); // can come from other include
#include "header.h"

void bar()
{
    func(); // `overloaded` refers to `void overloaded(char)`
}
```

Wir haben eine ODR-Verletzung, da `overloaded` abhängig von der Übersetzungseinheit auf verschiedene Entitäten verweist.

Eine Definitionsregel (ODR) online lesen: https://riptutorial.com/de/cplusplus/topic/4907/eine-definitionsregel--odr-

Kapitel 41: Einfädeln

Syntax

- Faden()
- Thread (Thread && Sonstiges)
- expliziter Thread (Funktion && func, Args && ... args)

Parameter

Parameter	Einzelheiten
`other`	Übernimmt den Besitz `other` , `other` besitzt den Thread nicht mehr
`func`	Funktion zum Aufrufen eines separaten Threads
`args`	Argumente für `func`

Bemerkungen

Einige Notizen:

- Zwei `std::thread` Objekte können **niemals** denselben Thread darstellen.
- Ein `std::thread` Objekt kann sich in einem Zustand befinden, in dem es **keinen** Thread darstellt (dh nach einer Verschiebung, nach dem Aufruf von `join` usw.).

Examples

Thread-Operationen

Wenn Sie einen Thread starten, wird er ausgeführt, bis er fertig ist.

An einem bestimmten Punkt müssen Sie (möglicherweise - der Thread ist möglicherweise bereits fertiggestellt) auf das Ende des Threads warten, da Sie beispielsweise das Ergebnis verwenden möchten.

```
int n;
std::thread thread{ calculateSomething, std::ref(n) };

//Doing some other stuff

//We need 'n' now!
//Wait for the thread to finish - if it is not already done
thread.join();

//Now 'n' has the result of the calculation done in the seperate thread
```

```
std::cout << n << '\n';
```

Sie können den Thread auch `detach` , damit er frei ausgeführt werden kann:

```
std::thread thread{ doSomething };

//Detaching the thread, we don't need it anymore (for whatever reason)
thread.detach();

//The thread will terminate when it is done, or when the main thread returns
```

Übergabe einer Referenz an einen Thread

Sie können einen Verweis (oder einen `const` Verweis) nicht direkt an einen Thread übergeben, da sie von `std::thread` kopiert / verschoben werden. Verwenden Sie stattdessen `std::reference_wrapper` :

```
void foo(int& b)
{
    b = 10;
}

int a = 1;
std::thread thread{ foo, std::ref(a) }; //'a' is now really passed as reference

thread.join();
std::cout << a << '\n'; //Outputs 10
```

```
void bar(const ComplexObject& co)
{
    co.doCalculations();
}

ComplexObject object;
std::thread thread{ bar, std::cref(object) }; //'object' is passed as const&

thread.join();
std::cout << object.getResult() << '\n'; //Outputs the result
```

Std :: thread erstellen

In C ++ werden Threads mit der Klasse std :: thread erstellt. Ein Thread ist ein separater Ausführungsablauf. Es ist analog zu einem Helfer, der eine Aufgabe ausführt, während Sie gleichzeitig eine andere ausführen. Wenn der gesamte Code im Thread ausgeführt wird, wird er *beendet* . Beim Erstellen eines Threads müssen Sie etwas übergeben, um darauf ausgeführt zu werden. Einige Dinge, die Sie an einen Thread übergeben können:

- Freie Funktionen
- Mitgliedfunktionen
- Functor-Objekte

- Lambda-Ausdrücke

Beispiel für eine freie Funktion - Führt eine Funktion in einem separaten Thread aus (Live-Beispiel):

```
#include <iostream>
#include <thread>

void foo(int a)
{
    std::cout << a << '\n';
}

int main()
{
    // Create and execute the thread
    std::thread thread(foo, 10); // foo is the function to execute, 10 is the
                                 // argument to pass to it

    // Keep going; the thread is executed separately

    // Wait for the thread to finish; we stay here until it is done
    thread.join();

    return 0;
}
```

Elementfunktion - führt eine Elementfunktion in einem separaten Thread aus (Live-Beispiel):

```
#include <iostream>
#include <thread>

class Bar
{
public:
    void foo(int a)
    {
        std::cout << a << '\n';
    }
};

int main()
{
    Bar bar;

    // Create and execute the thread
    std::thread thread(&Bar::foo, &bar, 10); // Pass 10 to member function

    // The member function will be executed in a separate thread

    // Wait for the thread to finish, this is a blocking operation
    thread.join();

    return 0;
}
```

Functor-Objektbeispiel (Live-Beispiel):

```
#include <iostream>
#include <thread>

class Bar
{
public:
    void operator()(int a)
    {
        std::cout << a << '\n';
    }
};

int main()
{
    Bar bar;

    // Create and execute the thread
    std::thread thread(bar, 10); // Pass 10 to functor object

    // The functor object will be executed in a separate thread

    // Wait for the thread to finish, this is a blocking operation
    thread.join();

    return 0;
}
```

Lambda-Ausdruckbeispiel (Live-Beispiel):

```
#include <iostream>
#include <thread>

int main()
{
    auto lambda = [](int a) { std::cout << a << '\n'; };

    // Create and execute the thread
    std::thread thread(lambda, 10); // Pass 10 to the lambda expression

    // The lambda expression will be executed in a separate thread

    // Wait for the thread to finish, this is a blocking operation
    thread.join();

    return 0;
}
```

Vorgänge im aktuellen Thread

`std::this_thread` ist ein `namespace` der Funktionen für den aktuellen Thread aus Funktionen enthält, aus denen er aufgerufen wird.

Funktion	Beschreibung
`get_id`	Gibt die ID des Threads zurück
`sleep_for`	Schläft für eine bestimmte Zeitspanne
`sleep_until`	Schläft bis zu einer bestimmten Zeit
`yield`	Planen Sie die Ausführung von Threads neu und geben Sie anderen Threads Priorität

---

`std::this_thread::get_id` der aktuellen Thread-ID mithilfe von `std::this_thread::get_id` :

```
void foo()
{
    //Print this threads id
    std::cout << std::this_thread::get_id() << '\n';
}

std::thread thread{ foo };
thread.join(); //'threads' id has now been printed, should be something like 12556

foo(); //The id of the main thread is printed, should be something like 2420
```

---

3 Sekunden lang schlafen mit `std::this_thread::sleep_for` :

```
void foo()
{
    std::this_thread::sleep_for(std::chrono::seconds(3));
}

std::thread thread{ foo };
foo.join();

std::cout << "Waited for 3 seconds!\n";
```

Mit `std::this_thread::sleep_until` bis 3 Stunden in der Zukunft `std::this_thread::sleep_until` :

```
void foo()
{
    std::this_thread::sleep_until(std::chrono::system_clock::now() + std::chrono::hours(3));
}

std::thread thread{ foo };
thread.join();

std::cout << "We are now located 3 hours after the thread has been called\n";
```

Lassen Sie andere Threads Priorität mit `std::this_thread::yield` :

```
void foo(int a)
```

```
{
    for (int i = 0; i < al ++i)
        std::this_thread::yield(); //Now other threads take priority, because this thread
                                   //isn't doing anything important

    std::cout << "Hello World!\n";
}

std::thread thread{ foo, 10 };
thread.join();
```

Verwenden von std :: async anstelle von std :: thread

`std::async` auch Threads `std::async` . Im Vergleich zu `std::thread` wird dies als weniger leistungsfähig betrachtet, aber einfacher zu verwenden, wenn Sie eine Funktion nur asynchron ausführen möchten.

Funktion asynchron aufrufen

```
#include <future>
#include <iostream>

unsigned int square(unsigned int i){
    return i*i;
}

int main() {
    auto f = std::async(std::launch::async, square, 8);
    std::cout << "square currently running\n"; //do something while square is running
    std::cout << "result is " << f.get() << '\n'; //getting the result from square
}
```

Häufige Fehler

- `std::async` gibt ein `std::future` , das den Rückgabewert enthält, der von der Funktion berechnet wird. Wenn diese `future` zerstört wird, wird gewartet, bis der Thread abgeschlossen ist, wodurch der Code effektiv zu einem Thread wird. Dies wird leicht übersehen, wenn Sie den Rückgabewert nicht benötigen:

  ```
  std::async(std::launch::async, square, 5);
  //thread already completed at this point, because the returning future got destroyed
  ```

- `std::async` funktioniert ohne `std::async` , also `std::async(square, 5);` kompiliert. Wenn Sie dies tun, kann das System entscheiden, ob es einen Thread erstellen möchte oder nicht. Die Idee war, dass das System einen Thread erstellt, es sei denn, es werden bereits mehr Threads ausgeführt, als effizient ausgeführt werden können. Leider entscheiden sich Implementierungen in der Regel einfach dafür, in dieser Situation keinen Thread zu erstellen. `std::launch::async` müssen Sie dieses Verhalten mit `std::launch::async` überschreiben, wodurch das System einen Thread erstellen muss.

- Hüte dich vor den Rennbedingungen.

Mehr über async bei Futures und Promises

Sicherstellen, dass ein Thread immer verbunden ist

Wenn der Destruktor für `std::thread` aufgerufen wird, um entweder ein Anruf `join()` oder `detach()` gemacht worden **sein.** Wenn ein Thread nicht verbunden oder getrennt wurde, wird standardmäßig `std::terminate` aufgerufen. Mit RAII ist dies im Allgemeinen einfach genug:

```
class thread_joiner
{
public:

    thread_joiner(std::thread t)
        : t_(std::move(t))
    { }

    ~thread_joiner()
    {
        if(t_.joinable()) {
            t_.join();
        }
    }

private:

    std::thread t_;
}
```

Dies wird dann wie folgt verwendet:

```
 void perform_work()
 {
     // Perform some work
 }

 void t()
 {
     thread_joiner j{std::thread(perform_work)};
     // Do some other calculations while thread is running
 } // Thread is automatically joined here
```

Dies bietet auch eine Ausnahmesicherheit. Wenn wir unseren Thread normalerweise erstellt hätten und die in `t()` Arbeit bei der Ausführung anderer Berechnungen eine Ausnahme ausgelöst hätte, wäre `join()` niemals für unseren Thread aufgerufen worden und unser Prozess wäre beendet worden.

Thread-Objekte neu zuordnen

Wir können leere Thread-Objekte erstellen und ihnen später Arbeit zuweisen.

Wenn wir ein `joinable` einem anderen aktiven, `joinable` Thread `joinable` , wird `std::terminate` automatisch aufgerufen, bevor der Thread ersetzt wird.

```
#include <thread>

void foo()
{
    std::this_thread::sleep_for(std::chrono::seconds(3));
}
//create 100 thread objects that do nothing
std::thread executors[100];

// Some code

// I want to create some threads now

for (int i = 0;i < 100;i++)
{
    // If this object doesn't have a thread assigned
    if (!executors[i].joinable())
         executors[i] = std::thread(foo);
}
```

Grundlegende Synchronisation

Thread-Synchronisierung kann unter Verwendung von Mutexen unter anderen Synchronisationsgrundelementen erreicht werden. Es gibt verschiedene Mutex-Typen, die von der Standardbibliothek bereitgestellt werden, aber der einfachste ist `std::mutex` . Um einen Mutex zu sperren, konstruieren Sie eine Sperre für ihn. Der einfachste Sperrentyp ist `std::lock_guard` :

```
std::mutex m;
void worker() {
    std::lock_guard<std::mutex> guard(m); // Acquires a lock on the mutex
    // Synchronized code here
} // the mutex is automatically released when guard goes out of scope
```

Bei `std::lock_guard` der Mutex für die gesamte Lebensdauer des `std::lock_guard` gesperrt. `std::unique_lock` Sie die Bereiche für das Sperren manuell steuern müssen, verwenden `std::unique_lock` stattdessen `std::unique_lock` :

```
std::mutex m;
void worker() {
    // by default, constructing a unique_lock from a mutex will lock the mutex
    // by passing the std::defer_lock as a second argument, we
    // can construct the guard in an unlocked state instead and
    // manually lock later.
    std::unique_lock<std::mutex> guard(m, std::defer_lock);
    // the mutex is not locked yet!
    guard.lock();
    // critical section
    guard.unlock();
    // mutex is again released
}
```

Weitere Thread-Synchronisationsstrukturen

Bedingungsvariablen verwenden

Eine Bedingungsvariable ist ein Grundelement, das in Verbindung mit einem Mutex verwendet wird, um die Kommunikation zwischen Threads zu orchestrieren. Dies ist zwar weder der ausschließliche noch der effizienteste Weg, aber für diejenigen, die mit dem Muster vertraut sind, kann es eines der einfachsten sein.

Man wartet auf eine `std::condition_variable` mit einem `std::unique_lock<std::mutex>` . Dadurch kann der Code den gemeinsam genutzten Zustand sicher prüfen, bevor er entscheidet, ob er mit der Erfassung fortfahren möchte oder nicht.

Nachfolgend finden Sie eine Hersteller-Konsumentenskizze, die `std::thread` , `std::condition_variable` , `std::mutex` und einige andere verwendet, um die Dinge interessant zu machen.

```
#include <condition_variable>
#include <cstddef>
#include <iostream>
#include <mutex>
#include <queue>
#include <random>
#include <thread>


int main()
{
    std::condition_variable cond;
    std::mutex mtx;
    std::queue<int> intq;
    bool stopped = false;

    std::thread producer{[&]()
    {
        // Prepare a random number generator.
        // Our producer will simply push random numbers to intq.
        //
        std::default_random_engine gen{};
        std::uniform_int_distribution<int> dist{};

        std::size_t count = 4006;
        while(count--)
        {
            // Always lock before changing
            // state guarded by a mutex and
            // condition_variable (a.k.a. "condvar").
            std::lock_guard<std::mutex> L{mtx};

            // Push a random int into the queue
            intq.push(dist(gen));

            // Tell the consumer it has an int
            cond.notify_one();
        }

        // All done.
        // Acquire the lock, set the stopped flag,
        // then inform the consumer.
        std::lock_guard<std::mutex> L{mtx};

        std::cout << "Producer is done!" << std::endl;
```

```
        stopped = true;
        cond.notify_one();
    }};

    std::thread consumer{[&]()
    {
        do{
            std::unique_lock<std::mutex> L{mtx};
            cond.wait(L,[&]()
            {
                // Acquire the lock only if
                // we've stopped or the queue
                // isn't empty
                return stopped || ! intq.empty();
            });

            // We own the mutex here; pop the queue
            // until it empties out.

            while( ! intq.empty())
            {
                const auto val = intq.front();
                intq.pop();

                std::cout << "Consumer popped: " << val << std::endl;
            }

            if(stopped){
                // producer has signaled a stop
                std::cout << "Consumer is done!" << std::endl;
                break;
            }

        }while(true);
    }};

    consumer.join();
    producer.join();

    std::cout << "Example Completed!" << std::endl;

    return 0;
}
```

Erstellen Sie einen einfachen Thread-Pool

C ++ 11-Threading-Grundelemente sind noch relativ niedrig. Sie können verwendet werden, um ein übergeordnetes Konstrukt wie einen Thread-Pool zu schreiben:

C ++ 14

```
struct tasks {
  // the mutex, condition variable and deque form a single
  // thread-safe triggered queue of tasks:
  std::mutex m;
  std::condition_variable v;
  // note that a packaged_task<void> can store a packaged_task<R>:
  std::deque<std::packaged_task<void()>> work;
```

```
// this holds futures representing the worker threads being done:
std::vector<std::future<void>> finished;

// queue( lambda ) will enqueue the lambda into the tasks for the threads
// to use.  A future of the type the lambda returns is given to let you get
// the result out.
template<class F, class R=std::result_of_t<F&()>>
std::future<R> queue(F&& f) {
  // wrap the function object into a packaged task, splitting
  // execution from the return value:
  std::packaged_task<R()> p(std::forward<F>(f));

  auto r=p.get_future(); // get the return value before we hand off the task
  {
    std::unique_lock<std::mutex> l(m);
    work.emplace_back(std::move(p)); // store the task<R()> as a task<void()>
  }
  v.notify_one(); // wake a thread to work on the task

  return r; // return the future result of the task
}

// start N threads in the thread pool.
void start(std::size_t N=1){
  for (std::size_t i = 0; i < N; ++i)
  {
    // each thread is a std::async running this->thread_task():
    finished.push_back(
      std::async(
        std::launch::async,
        [this]{ thread_task(); }
      )
    );
  }
}
// abort() cancels all non-started tasks, and tells every working thread
// stop running, and waits for them to finish up.
void abort() {
  cancel_pending();
  finish();
}
// cancel_pending() merely cancels all non-started tasks:
void cancel_pending() {
  std::unique_lock<std::mutex> l(m);
  work.clear();
}
// finish enques a "stop the thread" message for every thread, then waits for them:
void finish() {
  {
    std::unique_lock<std::mutex> l(m);
    for(auto&&unused:finished){
      work.push_back({});
    }
  }
  v.notify_all();
  finished.clear();
}
~tasks() {
  finish();
}
```

```
private:
  // the work that a worker thread does:
  void thread_task() {
    while(true){
      // pop a task off the queue:
      std::packaged_task<void()> f;
      {
        // usual thread-safe queue code:
        std::unique_lock<std::mutex> l(m);
        if (work.empty()){
          v.wait(l,[&]{return !work.empty();});
        }
        f = std::move(work.front());
        work.pop_front();
      }
      // if the task is invalid, it means we are asked to abort:
      if (!f.valid()) return;
      // otherwise, run the task:
      f();
    }
  }
};
```

`tasks.queue( []{ return "hello world"s; } )` gibt ein `std::future<std::string>` , das beim Ausführen des Aufgabenobjekts mit `hello world` .

Sie erstellen Threads, indem Sie `tasks.start(10)` (wodurch 10 Threads `tasks.start(10)`).

Die Verwendung von `packaged_task<void()>` ist lediglich darauf zurückzuführen, dass es keine mit dem Typ `std::function` äquivalente `std::function` , die Nur-Move-Typen speichert. Das Erstellen einer benutzerdefinierten Datei wäre wahrscheinlich schneller als die Verwendung von `packaged_task<void()>` .

Live-Beispiel .

C ++ 11

Ersetzen `result_of_t<blah>` in C ++ 11 `result_of_t<blah>` durch den `typename result_of<blah>::type` .

Mehr zu Mutexes .

Thread-lokaler Speicher

Thread-lokaler Speicher kann mit dem Schlüsselwort `thread_local` erstellt werden. Eine mit dem `thread_local` deklarierte Variable hat eine **Thread-Speicherdauer.**

- Jeder Thread in einem Programm verfügt über eine eigene Kopie jeder lokalen Threadvariablen.
- Eine lokale Thread-Variable mit dem Funktionsbereich (local) wird initialisiert, wenn die Steuerung zum ersten Mal ihre Definition durchläuft. Eine solche Variable ist implizit statisch, sofern sie nicht `extern` deklariert wird.
- Eine Thread-lokale Variable mit Namespace oder Klassenbereich (nicht lokal) wird beim

Start des Threads initialisiert.
- Threadlokale Variablen werden bei Beendigung des Threads zerstört.
- Ein Member einer Klasse kann nur dann threadlokal sein, wenn es statisch ist. Es gibt daher eine Kopie dieser Variablen pro Thread und nicht eine Kopie pro Paar (Thread, Instanz).

Beispiel:

```
void debug_counter() {
    thread_local int count = 0;
    Logger::log("This function has been called %d times by this thread", ++count);
}
```

Einfädeln online lesen: https://riptutorial.com/de/cplusplus/topic/699/einfadeln

Kapitel 42: Elision kopieren

Examples

Zweck der Auslassung von Kopien

Es gibt Stellen im Standard, an denen ein Objekt kopiert oder verschoben wird, um ein Objekt zu initialisieren. Copy elision (manchmal auch als Rückgabewertoptimierung bezeichnet) ist eine Optimierung, bei der ein Compiler unter bestimmten Umständen das Kopieren oder Verschieben vermeiden darf, obwohl der Standard dies für erforderlich hält.

Betrachten Sie die folgende Funktion:

```
std::string get_string()
{
  return std::string("I am a string.");
}
```

Gemäß dem strengen Wortlaut des Standards initialisiert diese Funktion einen temporären `std::string` , kopiert / verschiebt diesen in das Rückgabewertobjekt und zerstört dann das temporäre. Der Standard ist sehr klar, dass der Code auf diese Weise interpretiert wird.

Copy elision ist eine Regel, die es einem C ++ - Compiler erlaubt , die Erstellung des temporären Elements und die anschließende Kopie / Zerstörung zu *ignorieren* . Das heißt, der Compiler kann den Initialisierungsausdruck für das temporäre Element übernehmen und den Rückgabewert der Funktion direkt daraus initialisieren. Dies spart offensichtlich Leistung.

Es hat jedoch zwei sichtbare Auswirkungen auf den Benutzer:

1. Der Typ muss über den Copy / Move-Konstruktor verfügen, der aufgerufen worden wäre. Selbst wenn der Compiler das Kopieren / Verschieben aufhebt, muss der Typ kopiert / verschoben werden können.

2. Nebenwirkungen von Copy / Move-Konstruktoren können unter Umständen nicht garantiert werden, in denen eine Entscheidung getroffen werden kann. Folgendes berücksichtigen:

C ++ 11

```
struct my_type
{
  my_type() = default;
  my_type(const my_type &) {std::cout <<"Copying\n";}
  my_type(my_type &&) {std::cout <<"Moving\n";}
};

my_type func()
{
  return my_type();
}
```

Was wird der Aufruf von `func` tun? Nun, es wird niemals "Kopieren" gedruckt, da der temporäre Wert ein r- `my_type` ist und `my_type` ein beweglicher Typ ist. Wird es "Moving" drucken?

Ohne die Kopiereliminierungsregel müsste dies immer "Moving" gedruckt werden. Da die Kopierelisierungsregel jedoch existiert, kann der Bewegungskonstruktor aufgerufen werden oder nicht. es ist implementierungsabhängig.

Daher können Sie sich nicht auf den Aufruf von Copy / Move-Konstruktoren in Kontexten verlassen, in denen eine Entscheidung möglich ist.

Da elision eine Optimierung ist, unterstützt Ihr Compiler möglicherweise nicht in allen Fällen elision. Unabhängig davon, ob der Compiler einen bestimmten Fall auswählt oder nicht, muss der Typ die ausgeführte Operation trotzdem unterstützen. Wenn also eine Kopierkonstruktion entfernt wird, muss der Typ noch einen Kopierkonstruktor haben, auch wenn er nicht aufgerufen wird.

Garantierte Kopiereinstellung

C ++ 17

Normalerweise ist die Entscheidung eine Optimierung. Während praktisch jeder Compiler in den einfachsten Fällen eine Kopierentscheidung unterstützt, belasten die Benutzer dennoch die Elizierung. Der Typ, dessen Kopie / Verschiebung ausgesondert wird, *muss* immer noch über die Kopier- / Verschiebeoperation verfügen, die ausgelesen wurde.

Zum Beispiel:

```
std::mutex a_mutex;
std::lock_guard<std::mutex> get_lock()
{
  return std::lock_guard<std::mutex>(a_mutex);
}
```

Dies kann in Fällen nützlich sein, in denen `a_mutex` ein Mutex ist, der sich in einem privaten Besitz befindet, ein externer Benutzer jedoch eine Sperrung für bestimmte Bereiche wünschen möchte.

Dies ist auch nicht zulässig, da `std::lock_guard` nicht kopiert oder verschoben werden kann. Obwohl praktisch jeder C ++ - Compiler für das Kopieren / Verschieben zuständig ist, *muss* der Typ für diesen Typ verfügbar sein.

Bis C ++ 17.

In C ++ 17 müssen Sie die Eliminierung von Befehlen vornehmen, indem Sie die Bedeutung bestimmter Ausdrücke effektiv neu definieren, sodass kein Kopieren / Verschieben stattfindet. Betrachten Sie den obigen Code.

Bei der Formulierung vor C ++ 17 heißt es in diesem Code, eine temporäre Datei zu erstellen und dann die temporäre Kopie zum Kopieren / Verschieben in den Rückgabewert zu verwenden. Die temporäre Kopie kann jedoch entfernt werden. Unter dem Wortlaut von C ++ 17 wird dadurch überhaupt keine temporäre erstellt.

In C ++ 17 generiert ein Prvalue-Ausdruck bei der Initialisierung eines Objekts desselben Typs wie der Ausdruck kein temporäres Objekt. Der Ausdruck initialisiert das Objekt direkt. Wenn Sie einen pr-Wert desselben Typs zurückgeben, der dem Rückgabewert entspricht, muss der Typ keinen Kopier- / Verschiebungskonstruktor haben. Und unter den C ++ 17-Regeln kann der obige Code funktionieren.

Die Formulierung von C ++ 17 funktioniert in Fällen, in denen der Typ des prvalue mit dem zu initialisierenden Typ übereinstimmt. Bei gegebenem `get_lock` oben ist auch keine Kopie / Verschiebung erforderlich:

```
std::lock_guard the_lock = get_lock();
```

Da das Ergebnis von `get_lock` ein prvalue-Ausdruck ist, der zum Initialisieren eines Objekts desselben Typs verwendet wird, erfolgt kein Kopieren oder Verschieben. Dieser Ausdruck schafft niemals eine temporäre; es wird verwendet, um `the_lock` direkt zu initialisieren. Es gibt keine Entscheidung, weil es keine Kopie / Bewegung gibt, um elide elide zu sein.

Der Begriff "garantierte Kopienauswahl" ist daher etwas falsch. Dies ist jedoch der Name des Features, wie es für die C ++ 17-Standardisierung vorgeschlagen wird . Es garantiert überhaupt keine Ausscheidung; Dadurch *entfällt* das Kopieren / Verschieben insgesamt, *wodurch* C ++ neu definiert wird, so dass niemals ein Kopiervorgang ausgeführt wurde.

Diese Funktion funktioniert nur in Fällen, in denen ein Prvalue-Ausdruck enthalten ist. Als solches verwendet dies die üblichen Ausscheidungsregeln:

```
std::mutex a_mutex;
std::lock_guard<std::mutex> get_lock()
{
  std::lock_guard<std::mutex> my_lock(a_mutex);
  //Do stuff
  return my_lock;
}
```

Dies ist zwar ein gültiger Fall für das Kopieren von Kopien, aber C ++ 17-Regeln *beseitigen* in diesem Fall das Kopieren / Verschieben nicht. Daher muss der Typ noch einen Kopier- / Verschiebungskonstruktor enthalten, der zum Initialisieren des Rückgabewerts verwendet werden kann. Und da `lock_guard` dies nicht tut, ist dies immer noch ein Kompilierungsfehler. Implementierungen dürfen die Ausgabe von Kopien verweigern, wenn sie ein Objekt mit trivial kopierbarem Typ übergeben oder zurückgeben. Dies ermöglicht das Verschieben solcher Objekte in Registern, die einige ABIs möglicherweise in ihren Aufrufkonventionen festlegen.

```
struct trivially_copyable {
    int a;
};

void foo (trivially_copyable a) {}

foo(trivially_copyable{}); //copy elision not mandated
```

Rückgabewert elision

Wenn Sie einen prvalue-Ausdruck aus einer Funktion zurückgeben und der prvalue-Ausdruck denselben Typ wie der Rückgabetyp der Funktion hat, kann die Kopie aus dem temporären prvalue-Objekt entfernt werden:

```
std::string func()
{
  return std::string("foo");
}
```

So ziemlich alle Compiler werden in diesem Fall die temporäre Konstruktion übernehmen.

Parameterauswahl

Wenn Sie ein Argument an eine Funktion übergeben und das Argument ein prvalue-Ausdruck des Parametertyps der Funktion ist und dieser Typ keine Referenz ist, kann die Konstruktion des prvalue aufgehoben werden.

```
void func(std::string str) { ... }

func(std::string("foo"));
```

Dies bedeutet, dass Sie eine temporäre `string` erstellen und diese dann in den Funktionsparameter `str` . Copy elision erlaubt es diesem Ausdruck, das Objekt direkt in `str` zu erstellen, anstatt eine temporäre + Verschiebung zu verwenden.

Dies ist eine nützliche Optimierung für Fälle, in denen ein Konstruktor als `explicit` deklariert wird. Zum Beispiel hätten wir das Obige als `func("foo")` schreiben können, aber nur, weil `string` einen impliziten Konstruktor hat, der von `const char*` in einen `string` konvertiert. Wenn dieser Konstruktor `explicit` , müssen wir den `explicit` Konstruktor mit einem temporären Aufruf aufrufen. Copy elision erspart uns das unnötige Kopieren / Verschieben.

Benannte Rückgabewerte

Wenn Sie einen lvalue-Ausdruck von einer Funktion zurückgeben, und diesen lvalue:

- stellt eine automatische Variable lokal für diese Funktion dar, die nach der `return`
- Die automatische Variable ist kein Funktionsparameter
- und der Typ der Variablen ist derselbe Typ wie der Rückgabetyp der Funktion

Wenn dies alles der Fall ist, kann das Kopieren / Bewegen vom Wert abgelöst werden:

```
std::string func()
{
  std::string str("foo");
  //Do stuff
  return str;
}
```

Für komplexere Fälle kann eine Entscheidung getroffen werden, aber je komplexer der Fall ist, desto unwahrscheinlicher wird es sein, dass der Compiler die Entscheidung trifft.

```
std::string func()
{
  std::string ret("foo");
  if(some_condition)
  {
    return "bar";
  }
  return ret;
}
```

Der Compiler konnte noch elide `ret` , aber die Chancen, sie so tun , nach unten gehen.

Wie bereits erwähnt, ist für *Werteparameter* keine Entscheidung zulässig.

```
std::string func(std::string str)
{
  str.assign("foo");
  //Do stuff
  return str; //No elision possible
}
```

Kopieren Sie die Initialisierung

Wenn Sie zum Kopieren einer Variablen einen Prvalue-Ausdruck verwenden und diese Variable denselben Typ wie der Prvalue-Ausdruck hat, kann das Kopieren abgebrochen werden.

```
std::string str = std::string("foo");
```

Die Initialisierung beim Kopieren wandelt dies effektiv in `std::string str("foo");` (es gibt geringfügige Unterschiede).

Dies funktioniert auch mit Rückgabewerten:

```
std::string func()
{
  return std::string("foo");
}

std::string str = func();
```

Ohne Copy-Elision würde dies 2 Aufrufe des Move-Konstruktors von `std::string` hervorrufen. Die Kopierentscheidung erlaubt den Aufruf des Bewegungskonstruktors 1 oder null Mal, und die meisten Compiler entscheiden sich für Letzteres.

Elision kopieren online lesen: https://riptutorial.com/de/cplusplus/topic/2489/elision-kopieren

Kapitel 43: Explizite Typkonvertierungen

Einführung

Ein Ausdruck kann *explizit umgewandelt* oder *gegossen* werden auf den Typ `T` mit `dynamic_cast<T>` , `static_cast<T>` , `reinterpret_cast<T>` oder `const_cast<T>` , je nachdem , welche Art von Guss soll.

C ++ unterstützt auch Cast-Notation im Funktionsstil, `T(expr)` und Cast-Notation im C-Stil `(T)expr` .

Syntax

- *einfacher Typbezeichner* `(` `)`
- *einfacher Typbezeichner* `(` *Ausdrucksliste* `)`
- *Einfacher Typbezeichner Braced-Init-List*
- *Typname-Bezeichner* `(` `)`
- *Typname-Bezeichner* `(` *Ausdrucksliste* `)`
- *typename-spezifizierer braced -init-list*
- `dynamic_cast` `<` *Typ-ID* `>` `(` *Ausdruck* `)`
- `static_cast` `<` *Typ-ID* `>` `(` *Ausdruck* `)`
- `reinterpret_cast` `<` *Typ-ID* `>` `(` *Ausdruck* `)`
- `const_cast` `<` *Typ-ID* `>` `(` *Ausdruck* `)`
- `(` *Typ-ID* `)` *Cast-Ausdruck*

Bemerkungen

Alle sechs Cast-Notationen haben eines gemeinsam:

- Die Umwandlung in einen lvalue-Referenztyp, wie in `dynamic_cast<Derived&>(base)` , ergibt einen lvalue. Wenn Sie also mit demselben Objekt etwas tun, es aber als einen anderen Typ behandeln möchten, würden Sie es in einen Lvalue-Referenztyp umwandeln.
- Die Umwandlung in einen rvalue-Referenztyp, wie in `static_cast<string&&>(s)` , ergibt einen rvalue.
- Die Umwandlung in einen Nichtreferenztyp wie in `(int)x` ergibt einen pr-Wert, der als *Kopie* des gegossenen Werts betrachtet werden kann, jedoch mit einem anderen Typ als dem Original.

Das `reinterpret_cast` Schlüsselwort ist dafür verantwortlich, zwei verschiedene Arten von "unsicheren" Konvertierungen durchzuführen:

- Die "Typ-Punning" -Konvertierungen , die verwendet werden können, um auf den Speicher eines Typs zuzugreifen, als ob er von einem anderen Typ wäre.
- Konvertierungen zwischen Ganzzahl- und Zeigertypen in beide Richtungen.

Mit `static_cast` Schlüsselwort `static_cast` können verschiedene Konvertierungen durchgeführt werden:

- Basis für abgeleitete Konvertierungen
- Jede Konvertierung, die durch direkte Initialisierung durchgeführt werden kann, einschließlich impliziter Konvertierungen und Konvertierungen, die einen expliziten Konstruktor oder eine Konvertierungsfunktion aufrufen. Siehe hier und hier für weitere Details.
- `void` , wodurch der Wert des Ausdrucks verworfen wird.

```
// on some compilers, suppresses warning about x being unused
static_cast<void>(x);
```

- Zwischen arithmetischen und Aufzählungstypen und zwischen verschiedenen Aufzählungstypen. Siehe Enum-Konvertierungen
- Vom Zeiger zum Mitglied der abgeleiteten Klasse, zum Zeiger auf das Mitglied der Basisklasse. Die Typen, auf die gezeigt wird, müssen übereinstimmen. Siehe abgeleitete Basiskonvertierung für Zeiger auf Member
- `void*` bis `T*` .

C ++ 11

- Von einem lvalue zu einem xvalue wie in `std::move` . Siehe Verschiebesemantik .

Examples

Basis für abgeleitete Konvertierung

Ein Zeiger auf die Basisklasse kann mithilfe von `static_cast` in einen Zeiger auf eine abgeleitete Klasse `static_cast` . `static_cast` führt keine Laufzeitüberprüfung durch und kann zu undefiniertem Verhalten führen, wenn der Zeiger tatsächlich nicht auf den gewünschten Typ zeigt.

```
struct Base {};
struct Derived : Base {};
Derived d;
Base* p1 = &d;
Derived* p2 = p1;                          // error; cast required
Derived* p3 = static_cast<Derived*>(p1); // OK; p2 now points to Derived object
Base b;
Base* p4 = &b;
Derived* p5 = static_cast<Derived*>(p4); // undefined behaviour since p4 does not
                                           // point to a Derived object
```

Ebenso kann eine Referenz auf die Basisklasse mit `static_cast` in eine Referenz auf eine abgeleitete Klasse `static_cast` .

```
struct Base {};
struct Derived : Base {};
Derived d;
Base& r1 = d;
```

```
Derived& r2 = r1;                          // error; cast required
Derived& r3 = static_cast<Derived&>(r1); // OK; r3 now refers to Derived object
```

Wenn der `dynamic_cast` polymorph ist, kann mit `dynamic_cast` eine Konvertierung von Basis in abgeleitete `dynamic_cast` werden. Es führt eine Laufzeitprüfung durch und der Fehler kann behoben werden, anstatt undefiniertes Verhalten zu erzeugen. Im Zeigerfall wird bei einem Fehler ein Nullzeiger zurückgegeben. Im Referenzfall wird bei einem Fehler des Typs `std::bad_cast` (oder einer von `std::bad_cast` abgeleiteten Klasse) eine Ausnahme ausgelöst.

```
struct Base { virtual ~Base(); }; // Base is polymorphic
struct Derived : Base {};
Base* b1 = new Derived;
Derived* d1 = dynamic_cast<Derived*>(b1); // OK; d1 points to Derived object
Base* b2 = new Base;
Derived* d2 = dynamic_cast<Derived*>(b2); // d2 is a null pointer
```

Konstanz wegwerfen

Ein Zeiger auf ein const-Objekt kann mit dem Schlüsselwort `const_cast` in einen Zeiger auf ein Nicht-const-Objekt `const_cast` . Hier rufen wir `const_cast` auf, um eine Funktion aufzurufen, die nicht const-correct ist. Es akzeptiert nur ein nicht-konstantes `char*` -Argument, obwohl es niemals durch den Zeiger schreibt:

```
void bad_strlen(char*);
const char* s = "hello, world!";
bad_strlen(s);                     // compile error
bad_strlen(const_cast<char*>(s)); // OK, but it's better to make bad_strlen accept const char*
```

`const_cast` in reference type kann ein const-qualifizierter l-Wert in einen nicht-const-qualifizierten Wert umgewandelt werden.

`const_cast` ist gefährlich, da das C ++ - Typsystem Sie nicht daran hindern kann, ein const-Objekt zu ändern. Dies führt zu undefiniertem Verhalten.

```
const int x = 123;
int& mutable_x = const_cast<int&>(x);
mutable_x = 456; // may compile, but produces *undefined behavior*
```

Geben Sie Punning-Konvertierung ein

Ein Zeiger (bzw. eine Referenz) auf einen Objekttyp kann mit `reinterpret_cast` in einen Zeiger (bzw. eine Referenz) auf einen anderen Objekttyp konvertiert werden. Dies ruft keine Konstruktoren oder Konvertierungsfunktionen auf.

```
int x = 42;
char* p = static_cast<char*>(&x);      // error: static_cast cannot perform this conversion
char* p = reinterpret_cast<char*>(&x); // OK
*p = 'z';                              // maybe this modifies x (see below)
```

C ++ 11

Das Ergebnis von `reinterpret_cast` repräsentiert dieselbe Adresse wie der Operand, vorausgesetzt, die Adresse ist entsprechend dem Zieltyp ausgerichtet. Andernfalls ist das Ergebnis nicht spezifiziert.

```
int x = 42;
char& r = reinterpret_cast<char&>(x);
const void* px = &x;
const void* pr = &r;
assert(px == pr); // should never fire
```

C ++ 11

Das Ergebnis von `reinterpret_cast` ist nicht spezifiziert, mit der Ausnahme, dass ein Zeiger (bzw. eine Referenz) eine Rundreise vom Quelltyp zum Zieltyp und zurück überlebt, sofern die Ausrichtungsanforderung des Zieltyps nicht strenger ist als die des Quelltyps.

```
int x = 123;
unsigned int& r1 = reinterpret_cast<unsigned int&>(x);
int& r2 = reinterpret_cast<int&>(r1);
r2 = 456; // sets x to 456
```

Bei den meisten Implementierungen ändert `reinterpret_cast` die Adresse nicht, diese Anforderung wurde jedoch erst in C ++ 11 standardisiert.

`reinterpret_cast` kann auch verwendet werden, um einen Zeiger-zu-Daten-Mitgliedstyp in einen anderen oder einen Zeiger-zu-Mitglied-Funktionstyp in einen anderen umzuwandeln.

Die Verwendung von `reinterpret_cast` wird als gefährlich angesehen, da das Lesen oder Schreiben über einen Zeiger oder eine mit `reinterpret_cast` erhaltene Referenz ein undefiniertes Verhalten auslösen kann, wenn die Quell- und Zieltypen nicht miteinander zusammenhängen.

Konvertierung zwischen Zeiger und Ganzzahl

Ein Objektzeiger (einschließlich `void*`) oder Funktionszeiger kann mit `reinterpret_cast` in einen Integer-Typ konvertiert werden. Dies wird nur kompiliert, wenn der Zieltyp lang genug ist. Das Ergebnis ist implementierungsdefiniert und liefert normalerweise die numerische Adresse des Bytes im Speicher, auf das der Zeiger zeigt.

In der Regel ist `long` oder `unsigned long` lang genug, um einen Zeigerwert aufzunehmen, dies wird jedoch vom Standard nicht garantiert.

C ++ 11

Wenn die Typen `std::intptr_t` und `std::uintptr_t` vorhanden sind, sind sie garantiert lang genug, um ein `void*` (und somit einen Zeiger auf den Objekttyp) aufzunehmen. Sie sind jedoch nicht garantiert lang genug, um einen Funktionszeiger zu halten.

In ähnlicher Weise kann mit `reinterpret_cast` ein Integer-Typ in einen Zeigertyp konvertiert werden. Das Ergebnis ist wiederum implementierungsdefiniert, jedoch wird garantiert, dass ein Zeigerwert durch eine Rundreise durch einen Integer-Typ unverändert bleibt. Der Standard

garantiert nicht, dass der Wert Null in einen Nullzeiger umgewandelt wird.

```
void register_callback(void (*fp)(void*), void* arg); // probably a C API
void my_callback(void* x) {
    std::cout << "the value is: " << reinterpret_cast<long>(x); // will probably compile
}
long x;
std::cin >> x;
register_callback(my_callback,
                  reinterpret_cast<void*>(x)); // hopefully this doesn't lose information...
```

Konvertierung durch expliziten Konstruktor oder explizite Konvertierungsfunktion

Eine Konvertierung, die den Aufruf eines expliziten Konstruktors oder einer Konvertierungsfunktion beinhaltet, kann nicht implizit durchgeführt werden. Wir können die Konvertierung explizit mit `static_cast` . Die Bedeutung ist dieselbe wie bei einer direkten Initialisierung, außer dass das Ergebnis temporär ist.

```
class C {
    std::unique_ptr<int> p;
  public:
    explicit C(int* p) : p(p) {}
};
void f(C c);
void g(int* p) {
    f(p);                  // error: C::C(int*) is explicit
    f(static_cast<C>(p)); // ok
    f(C(p));               // equivalent to previous line
    C c(p); f(c);          // error: C is not copyable
}
```

Implizite Konvertierung

`static_cast` kann jede implizite Konvertierung durchführen. Diese Verwendung von `static_cast` kann gelegentlich nützlich sein, wie in den folgenden Beispielen:

- Bei der Übergabe von Argumenten an eine Ellipse ist der Argumenttyp "erwartet" nicht statisch bekannt, sodass keine implizite Konvertierung erfolgt.

  ```
  const double x = 3.14;
  printf("%d\n", static_cast<int>(x)); // prints 3
  // printf("%d\n", x); // undefined behaviour; printf is expecting an int here
  // alternative:
  // const int y = x; printf("%d\n", y);
  ```

  Ohne die explizite Typkonvertierung würde ein `double` Objekt an die Ellipse übergeben und ein undefiniertes Verhalten würde auftreten.

- Ein abgeleiteter Klassenzuweisungsoperator kann einen Basisklassenzuweisungsoperator wie folgt aufrufen:

```
struct Base { /* ... */ };
struct Derived : Base {
    Derived& operator=(const Derived& other) {
        static_cast<Base&>(*this) = other;
        // alternative:
        // Base& this_base_ref = *this; this_base_ref = other;
    }
};
```

Aufzählungsumwandlungen

`static_cast` kann von einem Integer- oder Fließkommatyp in einen Aufzählungstyp konvertieren (egal ob mit oder ohne Bereich) und *umgekehrt.* Es kann auch zwischen Aufzählungstypen konvertiert werden.

- Die Konvertierung von einem nicht begrenzten Aufzählungstyp in einen arithmetischen Typ ist eine implizite Konvertierung. Es ist möglich, aber nicht notwendig, `static_cast` zu verwenden.

C ++ 11

- Wenn ein Aufzählungstyp für Bereiche in einen arithmetischen Typ konvertiert wird:
    - Wenn der Wert der Enumeration genau im Zieltyp dargestellt werden kann, ist das Ergebnis dieser Wert.
    - Wenn der Zieltyp ein Integer-Typ ist, ist das Ergebnis sonst nicht angegeben.
    - Wenn der Zieltyp ein Gleitkommatyp ist, ist das Ergebnis dasselbe wie das Konvertieren in den zugrunde liegenden Typ und dann in den Gleitkommatyp.

  Beispiel:

```
enum class Format {
    TEXT = 0,
    PDF = 1000,
    OTHER = 2000,
};
Format f = Format::PDF;
int a = f;                          // error
int b = static_cast<int>(f);        // ok; b is 1000
char c = static_cast<char>(f);      // unspecified, if 1000 doesn't fit into char
double d = static_cast<double>(f); // d is 1000.0... probably
```

- Wenn eine Ganzzahl oder ein Aufzählungstyp in einen Aufzählungstyp konvertiert wird:
    - Wenn der ursprüngliche Wert innerhalb des Zielbereichs liegt, ist das Ergebnis dieser Wert. Beachten Sie, dass dieser Wert möglicherweise nicht allen Enumeratoren entspricht.
    - Andernfalls ist das Ergebnis nicht spezifiziert (<= C ++ 14) oder undefined (> = C ++ 17).

  Beispiel:

```
enum Scale {
    SINGLE = 1,
    DOUBLE = 2,
    QUAD = 4
};
Scale s1 = 1;                       // error
Scale s2 = static_cast<Scale>(2); // s2 is DOUBLE
Scale s3 = static_cast<Scale>(3); // s3 has value 3, and is not equal to any enumerator
Scale s9 = static_cast<Scale>(9); // unspecified value in C++14; UB in C++17
```

C ++ 11

- Wenn ein Gleitkommatyp in einen Aufzählungstyp konvertiert wird, entspricht das Ergebnis dem Konvertieren in den zugrunde liegenden Typ der Enumeration und dann in den Aufzählungstyp.

```
enum Direction {
    UP = 0,
    LEFT = 1,
    DOWN = 2,
    RIGHT = 3,
};
Direction d = static_cast<Direction>(3.14); // d is RIGHT
```

Abgeleitet in Basisumwandlung für Zeiger auf Mitglieder

Ein Zeiger auf ein Mitglied der abgeleiteten Klasse kann mit `static_cast` in einen Zeiger auf ein Mitglied der Basisklasse `static_cast` . Die Typen, auf die gezeigt wird, müssen übereinstimmen.

Wenn der Operand ein Nullzeiger auf den Elementwert ist, ist das Ergebnis auch ein Nullzeiger auf den Elementwert.

Andernfalls ist die Konvertierung nur gültig, wenn der Member, auf den der Operand zeigt, tatsächlich in der Zielklasse vorhanden ist oder wenn die Zielklasse eine Basis- oder abgeleitete Klasse der Klasse ist, die den Member enthält, auf den der Operand zeigt. `static_cast` nicht auf Gültigkeit. Wenn die Konvertierung nicht gültig ist, ist das Verhalten undefiniert.

```
struct A {};
struct B { int x; };
struct C : A, B { int y; double z; };
int B::*p1 = &B::x;
int C::*p2 = p1;                                // ok; implicit conversion
int B::*p3 = p2;                                // error
int B::*p4 = static_cast<int B::*>(p2);       // ok; p4 is equal to p1
int A::*p5 = static_cast<int A::*>(p2);       // undefined; p2 points to x, which is a member
                                                // of the unrelated class B
double C::*p6 = &C::z;
double A::*p7 = static_cast<double A::*>(p6); // ok, even though A doesn't contain z
int A::*p8 = static_cast<int A::*>(p6);       // error: types don't match
```

ungültig * bis T *

In C ++ kann `void*` nicht implizit in `T*` konvertiert werden, wobei `T` ein Objekttyp ist. Stattdessen

sollte `static_cast` verwendet werden, um die Konvertierung explizit durchzuführen. Wenn der Operand tatsächlich auf ein `T` Objekt zeigt, zeigt das Ergebnis auf dieses Objekt. Andernfalls ist das Ergebnis nicht spezifiziert.

C ++ 11

Selbst wenn der Operand nicht auf ein `T` Objekt zeigt, solange der Operand auf ein Byte zeigt, dessen Adresse für den Typ `T` richtig ausgerichtet ist, zeigt das Ergebnis der Konvertierung auf dasselbe Byte.

```
// allocating an array of 100 ints, the hard way
int* a = malloc(100*sizeof(*a));                    // error; malloc returns void*
int* a = static_cast<int*>(malloc(100*sizeof(*a))); // ok
// int* a = new int[100];                           // no cast needed
// std::vector<int> a(100);                         // better

const char c = '!';
const void* p1 = &c;
const char* p2 = p1;                               // error
const char* p3 = static_cast<const char*>(p1); // ok; p3 points to c
const int* p4 = static_cast<const int*>(p1);   // unspecified in C++03;
                                               // possibly unspecified in C++11 if
                                               // alignof(int) > alignof(char)
char* p5 = static_cast<char*>(p1);             // error: casting away constness
```

C-Style Casting

C-Style-Casting kann als "Best Effort" -Casting bezeichnet werden und wird so benannt, dass es das einzige Cast ist, das in C verwendet werden `(NewType)variable` .

Immer wenn dieser Cast verwendet wird, verwendet er einen der folgenden C ++ - Casts (in Reihenfolge):

- `const_cast<NewType>(variable)`
- `static_cast<NewType>(variable)`
- `const_cast<NewType>(static_cast<const NewType>(variable))`
- `reinterpret_cast<const NewType>(variable)`
- `const_cast<NewType>(reinterpret_cast<const NewType>(variable))`

Das funktionale Casting ist sehr ähnlich, jedoch mit einigen Einschränkungen aufgrund seiner Syntax: `NewType(expression)` . Daher können nur Typen ohne Leerzeichen umgewandelt werden.

Es ist besser, eine neue C ++ - Konvertierung zu verwenden, da sie besser lesbar ist und leicht an einem beliebigen Ort innerhalb eines C ++ - Quellcodes erkannt werden kann. Fehler werden zur Kompilierzeit statt zur Laufzeit erkannt.

Da diese Umwandlung zu einem unbeabsichtigten `reinterpret_cast` , wird dies häufig als gefährlich betrachtet.

Explizite Typkonvertierungen online lesen: https://riptutorial.com/de/cplusplus/topic/3090/explizite-typkonvertierungen

Kapitel 44: Fließkomma-Arithmetik

Examples

Fließkommazahlen sind komisch

Der erste Fehler, den fast jeder Programmierer begeht, setzt voraus, dass dieser Code wie beabsichtigt funktioniert:

```
float total = 0;
for(float a = 0; a != 2; a += 0.01f) {
    total += a;
}
```

Der Programmierer für Anfänger geht davon aus, dass dies jede einzelne Zahl im Bereich von `0, 0.01, 0.02, 0.03, ..., 1.97, 1.98, 1.99` aufsummiert und das Ergebnis `199` ergibt - die mathematisch korrekte Antwort.

Zwei Dinge passieren, die dies unwahr machen:

1. Das Programm, wie geschrieben, endet nie. `a` nie gleich `2` und die Schleife endet nie.
2. Wenn wir die Schleifenlogik neu schreiben, um `a < 2` zu überprüfen, wird die Schleife jedoch beendet, aber die Summe ist etwas anders als `199` . Auf IEEE754-kompatiblen Maschinen summiert sich diese Zahl häufig auf etwa `201` .

Der Grund dafür ist, dass **Gleitkommazahlen Näherungswerte ihrer zugewiesenen Werte darstellen** .

Das klassische Beispiel ist die folgende Berechnung:

```
double a = 0.1;
double b = 0.2;
double c = 0.3;
if(a + b == c)
    //This never prints on IEEE754-compliant machines
    std::cout << "This Computer is Magic!" << std::endl;
else
    std::cout << "This Computer is pretty normal, all things considered." << std::endl;
```

Der Programmierer sieht zwar drei in base10 geschriebene Zahlen, aber der Compiler (und die zugrunde liegende Hardware) sehen Binärzahlen. Da für `0.1` , `0.2` und `0.3` eine perfekte Division durch `10` erforderlich ist - was in einem Basis-10-System recht einfach, in einem Basis-2-System jedoch nicht möglich ist - müssen diese Zahlen in ungenauen Formaten gespeichert werden, ähnlich wie bei der Zahl `1/3` muss in der ungenauen Form `0.333333333333333...` in Base-10 gespeichert werden.

```
//64-bit floats have 53 digits of precision, including the whole-number-part.
double a =     0011111110111001100110011001100110011001100110011001100110011010; //imperfect
```

```
representation of 0.1
double b =     0011111111001001100110011001100110011001100110011001100110011010; //imperfect
representation of 0.2
double c =     0011111111010011001100110011001100110011001100110011001100110011; //imperfect
representation of 0.3
double a + b = 0011111111010011001100110011001100110011001100110011001100110100; //Note that
this is not quite equal to the "canonical" 0.3!
```

Fließkomma-Arithmetik online lesen: https://riptutorial.com/de/cplusplus/topic/5115/flie-komma-arithmetik

Kapitel 45: Freund-Schlüsselwort

Einführung

Gut konzipierte Klassen kapseln ihre Funktionalität ein, verbergen ihre Implementierung und bieten gleichzeitig eine saubere, dokumentierte Schnittstelle. Dies ermöglicht ein Redesign oder eine Änderung, solange die Benutzeroberfläche unverändert bleibt.

In einem komplexeren Szenario sind möglicherweise mehrere Klassen erforderlich, die auf den Implementierungsdetails der jeweils anderen basieren. Friend-Klassen und -Funktionen ermöglichen diesen Kollegen den Zugriff auf die Details der jeweils anderen, ohne die Verkapselung und das Ausblenden von Informationen der dokumentierten Schnittstelle zu beeinträchtigen.

Examples

Friend-Funktion

Eine Klasse oder Struktur kann jede Funktion deklarieren, die sie als Freund bezeichnet. Wenn eine Funktion ein Freund einer Klasse ist, kann sie auf alle geschützten und privaten Mitglieder zugreifen:

```
// Forward declaration of functions.
void friend_function();
void non_friend_function();

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
private:
    int private_value;
    // Declare one of the function as a friend.
    friend void friend_function();
};

void non_friend_function() {
    PrivateHolder ph(10);
    // Compilation error: private_value is private.
    std::cout << ph.private_value << std::endl;
}

void friend_function() {
    // OK: friends may access private values.
    PrivateHolder ph(10);
    std::cout << ph.private_value << std::endl;
}
```

Zugriffsmodifizierer ändern die Semantik von Freunden nicht. Öffentliche, geschützte und private Erklärungen eines Freundes sind gleichwertig.

Freunderklärungen werden nicht vererbt. Wenn wir beispielsweise `PrivateHolder` subclass:

```
class PrivateHolderDerived : public PrivateHolder {
public:
    PrivateHolderDerived(int val) : PrivateHolder(val) {}
private:
    int derived_private_value = 0;
};
```

und versuchen, auf seine Mitglieder zuzugreifen, erhalten wir Folgendes:

```
void friend_function() {
    PrivateHolderDerived pd(20);
    // OK.
    std::cout << pd.private_value << std::endl;
    // Compilation error: derived_private_value is private.
    std::cout << pd.derived_private_value << std::endl;
}
```

Beachten Sie, dass `PrivateHolderDerived` Member-Funktion von `PrivateHolderDerived` nicht auf `PrivateHolder::private_value` zugreifen `PrivateHolder::private_value` , während die friend-Funktion dies tun kann.

Friend-Methode

Methoden können sowohl als Freunde als auch als Funktionen deklariert werden:

```
class Accesser {
public:
    void private_accesser();
};

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    friend void Accesser::private_accesser();
private:
    int private_value;
};

void Accesser::private_accesser() {
    PrivateHolder ph(10);
    // OK: this method is declares as friend.
    std::cout << ph.private_value << std::endl;
}
```

Freundenklasse

Eine ganze Klasse kann als Freund deklariert werden. Freundesklassenerklärung bedeutet, dass jedes Mitglied des Freundes auf private und geschützte Mitglieder der deklarierenden Klasse zugreifen kann:

```
class Accesser {
```

```
public:
    void private_accesser1();
    void private_accesser2();
};

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    friend class Accesser;
private:
    int private_value;
};

void Accesser::private_accesser1() {
    PrivateHolder ph(10);
    // OK.
    std::cout << ph.private_value << std::endl;
}

void Accesser::private_accesser2() {
    PrivateHolder ph(10);
    // OK.
    std::cout << ph.private_value + 1 << std::endl;
}
```

Die Erklärung der Freundschaftsklasse ist nicht reflexiv. Wenn Klassen in beiden Richtungen einen privaten Zugriff benötigen, benötigen beide eine Freundeserklärung.

```
class Accesser {
public:
    void private_accesser1();
    void private_accesser2();
private:
    int private_value = 0;
};

class PrivateHolder {
public:
    PrivateHolder(int val) : private_value(val) {}
    // Accesser is a friend of PrivateHolder
    friend class Accesser;
    void reverse_accesse() {
        // but PrivateHolder cannot access Accesser's members.
        Accesser a;
        std::cout << a.private_value;
    }
private:
    int private_value;
};
```

Freund-Schlüsselwort online lesen: https://riptutorial.com/de/cplusplus/topic/3275/freund-schlusselwort

Kapitel 46: Funktionsüberladung

Einführung

Siehe auch separates Thema zur Überlastauflösung

Bemerkungen

Mehrdeutigkeiten können auftreten, wenn ein Typ implizit in mehr als einen Typ konvertiert werden kann und für diesen bestimmten Typ keine übereinstimmende Funktion vorhanden ist.

Zum Beispiel:

```
void foo(double, double);
void foo(long, long);

//Call foo with 2 ints
foo(1, 2); //Function call is ambiguous - int can be converted into a double/long at the same time
```

Examples

Was ist Funktionsüberladung?

Beim Überladen von Funktionen werden mehrere Funktionen im selben Gültigkeitsbereich deklariert, wobei derselbe Name an derselben Stelle (bekannt als *Gültigkeitsbereich*) vorhanden ist und sich nur in der *Signatur unterscheidet* , dh die von ihnen akzeptierten Argumente.

Angenommen, Sie schreiben eine Reihe von Funktionen für allgemeine Druckfunktionen, beginnend mit `std::string` :

```
void print(const std::string &str)
{
    std::cout << "This is a string: " << str << std::endl;
}
```

Das funktioniert gut, aber Sie wollen eine Funktion, die auch ein `int` akzeptiert und das auch druckt. Sie könnten schreiben:

```
void print_int(int num)
{
    std::cout << "This is an int:  " << num << std::endl;
}
```

Da die beiden Funktionen jedoch unterschiedliche Parameter akzeptieren, können Sie einfach schreiben:

```
void print(int num)
{
    std::cout << "This is an int: " << num << std::endl;
}
```

Jetzt haben Sie 2 Funktionen, beide mit dem Namen `print` , jedoch mit unterschiedlichen Signaturen. Einer akzeptiert `std::string` , der andere ein `int` . Jetzt können Sie sie anrufen, ohne sich um verschiedene Namen zu kümmern:

```
print("Hello world!"); //prints "This is a string: Hello world!"
print(1337);           //prints "This is an int: 1337"
```

Anstatt:

```
print("Hello world!");
print_int(1337);
```

Wenn Sie Funktionen überladen haben, ermittelt der Compiler, welche der Funktionen von den von Ihnen angegebenen Parametern aufgerufen werden soll. Beim Schreiben von Funktionsüberladungen ist Vorsicht geboten. Zum Beispiel bei impliziten Typkonvertierungen:

```
void print(int num)
{
    std::cout << "This is an int: " << num << std::endl;
}
void print(double num)
{
    std::cout << "This is a double: " << num << std::endl;
}
```

Nun ist nicht sofort klar, welche Überladung des `print` beim Schreiben aufgerufen wird:

```
print(5);
```

Möglicherweise müssen Sie Ihrem Compiler einige Hinweise geben, z.

```
print(static_cast<double>(5));
print(static_cast<int>(5));
print(5.0);
```

Beim Schreiben von Überladungen, die optionale Parameter akzeptieren, ist auch etwas Vorsicht geboten:

```
// WRONG CODE
void print(int num1, int num2 = 0)    //num2 defaults to 0 if not included
{
    std::cout << "These are ints: << num1 << " and " << num2 << std::endl;
}
void print(int num)
{
    std::cout << "This is an int: " << num << std::endl;
}
```

Da der Compiler aufgrund des optionalen zweiten Parameters nicht feststellen kann, ob ein Aufruf wie `print(17)` für die erste oder zweite Funktion bestimmt ist, wird der Compiler nicht kompiliert.

Rückgabetyp bei Überladen der Funktion

Beachten Sie, dass Sie eine Funktion nicht basierend auf ihrem Rückgabetyp überladen können. Zum Beispiel:

```
// WRONG CODE
std::string getValue()
{
  return "hello";
}

int getValue()
{
  return 0;
}

int x = getValue();
```

Dies verursacht einen Kompilierungsfehler, da der Compiler nicht herausfinden kann, welche Version von `getValue` aufgerufen werden soll, obwohl der Rückgabetyp einem `int` zugewiesen ist.

Member Function cv-qualifier Überladung

Funktionen innerhalb einer Klasse können überlastet werden, wenn auf sie über eine von cv qualifizierte Referenz auf diese Klasse zugegriffen wird. Dies wird am häufigsten für Überladungen von `const` , kann aber auch für `volatile` und `const volatile` werden. Dies liegt daran, dass alle nicht statischen Elementfunktionen `this` verborgenen Parameter annehmen, auf den die cv-qualifiers angewendet werden. Dies wird am häufigsten für Überladung von `const` , kann aber auch für `volatile` und `const volatile` .

Dies ist notwendig, da eine Memberfunktion nur aufgerufen werden kann, wenn sie mindestens so hoch ist wie die Instanz, für die sie aufgerufen wird. Während eine Nicht- `const` Instanz sowohl `const` als auch nicht- `const` Member aufrufen kann, kann eine `const` Instanz nur `const` Member aufrufen. Dadurch kann eine Funktion je nach den CV-Qualifikationsmerkmalen der aufrufenden Instanz ein unterschiedliches Verhalten aufweisen, und der Programmierer kann Funktionen für unerwünschte CV-Qualifikationsmerkmale verbieten, indem keine Version mit diesen Qualifikationsmerkmalen bereitgestellt wird.

Eine Klasse mit einigen grundlegenden `print` könnte `const` wie so überlastet:

```
#include <iostream>

class Integer
{
    public:
        Integer(int i_): i{i_}{}

        void print()
        {
```

```
            std::cout << "int: " << i << std::endl;
        }

        void print() const
        {
            std::cout << "const int: " << i << std::endl;
        }

    protected:
        int i;
};

int main()
{
    Integer i{5};
    const Integer &ic = i;

    i.print(); // prints "int: 5"
    ic.print(); // prints "const int: 5"
}
```

Dies ist ein wesentlicher Grundsatz der `const` Korrektheit: Durch das Markieren von Member-Funktionen als `const` können sie für `const` Instanzen aufgerufen werden, `const` Funktionen Instanzen als `const` Zeiger / Referenzen verwenden können, wenn sie nicht geändert werden müssen. Dadurch kann der Code angeben, ob er den Status ändert, indem er nicht modifizierte Parameter als `const` und modifizierte Parameter ohne cv-qualifiers verwendet, wodurch Code sicherer und lesbarer wird.

```
class ConstCorrect
{
  public:
    void good_func() const
    {
        std::cout << "I care not whether the instance is const." << std::endl;
    }

    void bad_func()
    {
        std::cout << "I can only be called on non-const, non-volatile instances." <<
std::endl;
    }
};

void i_change_no_state(const ConstCorrect& cc)
{
    std::cout << "I can take either a const or a non-const ConstCorrect." << std::endl;
    cc.good_func(); // Good.  Can be called from const or non-const instance.
    cc.bad_func();  // Error.  Can only be called from non-const instance.
}

void const_incorrect_func(ConstCorrect& cc)
{
    cc.good_func(); // Good.  Can be called from const or non-const instance.
    cc.bad_func();  // Good.  Can only be called from non-const instance.
}
```

Eine gemeinsame Nutzung hierfür ist Accessoren als erklärt `const` und Mutatoren als nicht `const` .

Innerhalb einer `const` können keine Klassenmitglieder geändert werden. Wenn es ein Mitglied gibt, das Sie wirklich ändern müssen, z. B. das Sperren eines `std::mutex` , können Sie es als `mutable` deklarieren:

```
class Integer
{
    public:
        Integer(int i_): i{i_}{}

        int get() const
        {
            std::lock_guard<std::mutex> lock{mut};
            return i;
        }

        void set(int i_)
        {
            std::lock_guard<std::mutex> lock{mut};
            i = i_;
        }

    protected:
        int i;
        mutable std::mutex mut;
};
```

Funktionsüberladung online lesen:
https://riptutorial.com/de/cplusplus/topic/510/funktionsuberladung

Kapitel 47: Funktionsvorlage überladen

Bemerkungen

- Eine normale Funktion bezieht sich niemals auf eine Funktionsvorlage, obwohl sie denselben Namen und denselben Typ hat.
- Ein normaler Funktionsaufruf und ein generierter Funktionsvorlagenaufruf unterscheiden sich, auch wenn sie denselben Namen, denselben Rückgabetyp und dieselbe Argumentliste haben

Examples

Was ist das Überladen einer gültigen Funktionsvorlage?

Eine Funktionsvorlage kann nach den Regeln für das Überladen von Nicht-Vorlagenfunktionen (gleicher Name, aber unterschiedliche Parametertypen) überladen werden. Darüber hinaus gilt das Überladen, wenn

- Der Rückgabetyp ist anders oder
- Die Vorlagenparameterliste unterscheidet sich, außer der Benennung von Parametern und dem Vorhandensein von Standardargumenten (sie sind nicht Teil der Signatur).

Für eine normale Funktion ist der Vergleich zweier Parametertypen für den Compiler einfach, da er alle Informationen enthält. Ein Typ innerhalb einer Vorlage kann jedoch noch nicht bestimmt werden. Daher ist die Regel, wenn zwei Parametertypen gleich sind, hier annähernd und besagt, dass die nicht abhängigen Typen und Werte übereinstimmen müssen und die Schreibweise der abhängigen Typen und Ausdrücke gleich sein muss (genauer gesagt, sie müssen dem entsprechen sog. ODR-Regeln), mit der Ausnahme, dass Vorlagenparameter umbenannt werden können. Wenn jedoch unter solchen unterschiedlichen Schreibweisen zwei Werte innerhalb der Typen als unterschiedlich angesehen werden, jedoch immer die gleichen Werte vorhanden sind, ist die Überladung ungültig, aber vom Compiler ist keine Diagnose erforderlich.

```
template<typename T>
void f(T*) { }

template<typename T>
void f(T) { }
```

Dies ist eine gültige Überladung, da "T" und "T *" unterschiedliche Schreibweisen sind. Folgendes ist jedoch ungültig, es ist keine Diagnose erforderlich

```
template<typename T>
void f(T (*x)[sizeof(T) + sizeof(T)]) { }

template<typename T>
void f(T (*x)[2 * sizeof(T)]) { }
```

Funktionsvorlage überladen online lesen:
https://riptutorial.com/de/cplusplus/topic/4164/funktionsvorlage-uberladen

Kapitel 48: Futures und Versprechen

Einführung

Versprechen und Futures werden verwendet, um ein einzelnes Objekt von einem Thread zum anderen zu transportieren.

Ein `std::promise` Objekt wird vom Thread festgelegt, der das Ergebnis generiert.

Ein `std::future` Objekt kann verwendet werden, um einen Wert abzurufen, um zu testen, ob ein Wert verfügbar ist, oder um die Ausführung anzuhalten, bis der Wert verfügbar ist.

Examples

std :: future und std :: versprechen

Im folgenden Beispiel wird ein Versprechen für einen anderen Thread festgelegt:

```
    {
        auto promise = std::promise<std::string>();

        auto producer = std::thread([&]
        {
            promise.set_value("Hello World");
        });

        auto future = promise.get_future();

        auto consumer = std::thread([&]
        {
            std::cout << future.get();
        });

        producer.join();
        consumer.join();
}
```

Verzögertes asynchrones Beispiel

Dieser Code implementiert eine Version von `std::async` , verhält sich jedoch so, als würde `async` immer mit der `deferred async` aufgerufen. Diese Funktion hat auch kein besonderes `future` Verhalten von `async` . Die zurückgegebene `future` kann zerstört werden, ohne jemals ihren Wert zu erlangen.

```
template<typename F>
auto async_deferred(F&& func) -> std::future<decltype(func())>
{
    using result_type = decltype(func());

    auto promise = std::promise<result_type>();
```

```
    auto future  = promise.get_future();

    std::thread(std::bind([=](std::promise<result_type>& promise)
    {
        try
        {
            promise.set_value(func());
            // Note: Will not work with std::promise<void>. Needs some meta-template
programming which is out of scope for this example.
        }
        catch(...)
        {
            promise.set_exception(std::current_exception());
        }
    }, std::move(promise))).detach();

    return future;
}
```

std :: packaged_task und std :: future

`std::packaged_task` bündelt eine Funktion und das zugehörige Versprechen für den Rückgabetyp:

```
template<typename F>
auto async_deferred(F&& func) -> std::future<decltype(func())>
{
    auto task   = std::packaged_task<decltype(func())()>(std::forward<F>(func));
    auto future = task.get_future();

    std::thread(std::move(task)).detach();

    return std::move(future);
}
```

Der Thread läuft sofort an. Wir können es entweder entfernen oder am Ende des Bereichs hinzufügen. Wenn der Funktionsaufruf von std :: thread beendet ist, ist das Ergebnis fertig.

Beachten Sie, dass sich dies geringfügig von `std::async` wo das zurückgegebene `std::future` bei der Zerstörung tatsächlich **blockiert wird,** bis der Thread beendet ist.

std :: future_error und std :: future_errc

Wenn Einschränkungen für std :: promise und std :: future nicht erfüllt sind, wird eine Ausnahme vom Typ std :: future_error ausgelöst.

Der Fehlercode-Member in der Ausnahme ist vom Typ std :: future_errc. Die Werte lauten wie folgt und einige Testfälle:

```
enum class future_errc {
    broken_promise             = /* the task is no longer shared */,
    future_already_retrieved   = /* the answer was already retrieved */,
    promise_already_satisfied  = /* the answer was stored already */,
    no_state                   = /* access to a promise in non-shared state */
};
```

Inaktives Versprechen:

```
int test()
{
    std::promise<int> pr;
    return 0; // returns ok
}
```

Aktives Versprechen, nicht genutzt:

```
  int test()
    {
        std::promise<int> pr;
        auto fut = pr.get_future(); //blocks indefinitely!
        return 0;
    }
```

Doppelabruf:

```
int test()
{
    std::promise<int> pr;
    auto fut1 = pr.get_future();

    try{
        auto fut2 = pr.get_future();    //   second attempt to get future
        return 0;
    }
    catch(const std::future_error& e)
    {
        cout << e.what() << endl;       //   Error: "The future has already been retrieved
from the promise or packaged_task."
        return -1;
    }
    return fut2.get();
}
```

Std :: Versprechen zweimal einstellen:

```
int test()
{
    std::promise<int> pr;
    auto fut = pr.get_future();
    try{
        std::promise<int> pr2(std::move(pr));
        pr2.set_value(10);
        pr2.set_value(10);  // second attempt to set promise throws exception
    }
    catch(const std::future_error& e)
    {
        cout << e.what() << endl;       //   Error: "The state of the promise has already been
set."
        return -1;
    }
    return fut.get();
}
```

std :: future und std :: async

Im folgenden naiven Parallel Merge Sort-Beispiel wird `std::async` zum Starten mehrerer paralleler merge_sort-Tasks verwendet. `std::future` wird verwendet, um auf die Ergebnisse zu warten und sie zu synchronisieren:

```
#include <iostream>
using namespace std;


void merge(int low,int mid,int high, vector<int>&num)
{
    vector<int> copy(num.size());
    int h,i,j,k;
    h=low;
    i=low;
    j=mid+1;

    while((h<=mid)&&(j<=high))
    {
        if(num[h]<=num[j])
        {
            copy[i]=num[h];
            h++;
        }
        else
        {
            copy[i]=num[j];
            j++;
        }
        i++;
    }
    if(h>mid)
    {
        for(k=j;k<=high;k++)
        {
            copy[i]=num[k];
            i++;
        }
    }
    else
    {
        for(k=h;k<=mid;k++)
        {
            copy[i]=num[k];
            i++;
        }
    }
    for(k=low;k<=high;k++)
        swap(num[k],copy[k]);
}


void merge_sort(int low,int high,vector<int>& num)
{
    int mid;
    if(low<high)
    {
        mid = low + (high-low)/2;
```

```
        auto future1    =  std::async(std::launch::deferred,[&]()
                                      {
                                        merge_sort(low,mid,num);
                                      });
        auto future2    =  std::async(std::launch::deferred, [&]()
                                       {
                                          merge_sort(mid+1,high,num) ;
                                       });

        future1.get();
        future2.get();
        merge(low,mid,high,num);
    }
}
```

Hinweis: Im Beispiel wird `std::async` mit der Richtlinie `std::launch_deferred` . Dadurch wird vermieden, dass bei jedem Aufruf ein neuer Thread erstellt wird. In unserem Beispiel sind die Aufrufe von `std::async` nicht in der `std::async` Reihenfolge, sie synchronisieren sich bei den Aufrufen von `std::future::get()` .

`std::launch_async` bei jedem Aufruf ein neuer Thread erstellt.

Die Standardrichtlinie lautet `std::launch::deferred| std::launch::async` , dh die Implementierung bestimmt die Richtlinie zum Erstellen neuer Threads.

Async-Operationsklassen

- std :: async: führt eine asynchrone Operation aus.
- std :: future: ermöglicht den Zugriff auf das Ergebnis einer asynchronen Operation.
- std :: promise: packt das Ergebnis einer asynchronen Operation.
- std :: packaged_task: bündelt eine Funktion und das zugehörige Versprechen für den Rückgabetyp.

Futures und Versprechen online lesen: https://riptutorial.com/de/cplusplus/topic/9840/futures-und-versprechen

Kapitel 49: Geben Sie Schlüsselwörter ein

Examples

Klasse

1. Führt die Definition eines Klassentyps ein .

```
class foo {
    int x;
  public:
    int get_x();
    void set_x(int new_x);
};
```

2. Führt einen *ausführlichen Typbezeichner ein,* der angibt, dass der folgende Name der Name eines Klassentyps ist. Wenn der Klassenname bereits deklariert wurde, kann er auch gefunden werden, wenn er von einem anderen Namen ausgeblendet wird. Wenn der Klassenname noch nicht deklariert wurde, wird er vorwärts deklariert.

```
class foo; // elaborated type specifier -> forward declaration
class bar {
  public:
    bar(foo& f);
};
void baz();
class baz; // another elaborated type specifer; another forward declaration
           // note: the class has the same name as the function void baz()
class foo {
    bar b;
    friend class baz; // elaborated type specifier refers to the class,
                      // not the function of the same name
  public:
    foo();
};
```

3. Führt einen Typparameter in die Deklaration einer Vorlage ein .

```
template <class T>
const T& min(const T& x, const T& y) {
    return b < a ? b : a;
}
```

4. In der Deklaration eines Template - `class` Template - Parameter , das Schlüsselwort `class` steht vor dem Namen des Parameters. Da das Argument für einen Vorlagenvorlagenparameter nur eine Klassenvorlage sein kann, ist die Verwendung der `class` hier überflüssig. Die Grammatik von C ++ erfordert es jedoch.

```
template <template <class T> class U>
//                          ^^^^^ "class" used in this sense here;
```

```
//                                  U is a template template parameter
void f() {
    U<int>::do_it();
    U<double>::do_it();
}
```

5. Beachten Sie, dass Sinn 2 und Sinn 3 in derselben Deklaration zusammengefasst werden können. Zum Beispiel:

```
template <class T>
class foo {
};

foo<class bar> x; // <- bar does not have to have previously appeared.
```

C ++ 11

6. In der Deklaration oder Definition einer Aufzählung wird die Aufzählung als eine Aufzählung definiert .

```
enum class Format {
    TEXT,
    PDF,
    OTHER,
};
Format f = F::TEXT;
```

struct

Mit `class` austauschbar, mit Ausnahme der folgenden Unterschiede:

- Wenn ein Klassentyp mit dem Schlüsselwort `struct` , ist der Standardzugriff für Basen und Member `public` und nicht `private` .
- `struct` kann nicht verwendet werden, um einen Vorlagentypparameter oder einen Vorlagenvorlagenparameter zu deklarieren. Nur die `class` kann.

enum

1. Führt die Definition eines Aufzählungstyps ein .

```
enum Direction {
    UP,
    LEFT,
    DOWN,
    RIGHT
};
Direction d = UP;
```

C ++ 11

In C ++ 11 kann der `enum` optional eine `class` oder eine `struct` folgen, um eine Aufzählung mit

Bereich zu definieren. Darüber hinaus kann der zugrunde liegende Typ sowohl für bereichs- als auch für nicht begrenzte Enums explizit angegeben werden durch `: T` nach dem Namen der Enummen, wobei `T` auf einen Integer-Typ verweist.

```
enum class Format : char {
    TEXT,
    PDF,
    OTHER
};
Format f = Format::TEXT;

enum Language : int {
    ENGLISH,
    FRENCH,
    OTHER
};
```

Enumeratoren in normalen `enum` können auch vom Bereichsoperator vorangestellt werden, obwohl sie immer noch in dem Bereich liegen, in dem die `enum` definiert wurde.

```
Language l1, l2;

l1 = ENGLISH;
l2 = Language::OTHER;
```

2. Führt einen *ausführlichen Typbezeichner ein,* der angibt, dass der folgende Name der Name eines zuvor deklarierten Aufzählungstyps ist. (Ein ausführlicher Typbezeichner kann nicht zur Vorwärtsdeklaration eines Aufzählungstyps verwendet werden.) Eine Aufzählung kann auf diese Weise auch dann benannt werden, wenn sie von einem anderen Namen verborgen wird.

   ```
   enum Foo { FOO };
   void Foo() {}
   Foo foo = FOO;      // ill-formed; Foo refers to the function
   enum Foo foo = FOO; // ok; Foo refers to the enum type
   ```

C ++ 11

3. Führt eine *opake Enum-Deklaration ein,* die eine Enumeration deklariert, ohne sie zu definieren. Es kann entweder eine zuvor deklarierte Aufzählung erneut deklarieren oder eine zuvor deklarierte Aufzählung vorwärts deklarieren.

   Eine Enumeration, die zuerst als Gültigkeitsbereich deklariert wurde, kann später nicht als unbegrenzt deklariert werden oder *umgekehrt.* Alle Deklarationen einer Aufzählung müssen im zugrunde liegenden Typ übereinstimmen.

   Bei der Vorwärtsdeklaration eines unscoped-Enums muss der zugrunde liegende Typ explizit angegeben werden, da er nicht abgeleitet werden kann, bis die Werte der Enumeratoren bekannt sind.

   ```
   enum class Format; // underlying type is implicitly int
   ```

```
void f(Format f);
enum class Format {
    TEXT,
    PDF,
    OTHER,
};

enum Direction;    // ill-formed; must specify underlying type
```

Union

1. Führt die Definition eines Unionstyps ein .

```
// Example is from POSIX
union sigval {
    int     sival_int;
    void   *sival_ptr;
};
```

2. Führt einen *ausführlichen Typbezeichner ein,* der angibt, dass der folgende Name der Name eines Unionstyps ist. Wenn der Unionsname bereits deklariert wurde, kann er auch gefunden werden, wenn er von einem anderen Namen verborgen wird. Wenn der Unionsname noch nicht deklariert wurde, wird er vorwärts deklariert.

```
union foo; // elaborated type specifier -> forward declaration
class bar {
  public:
    bar(foo& f);
};
void baz();
union baz; // another elaborated type specifer; another forward declaration
           // note: the class has the same name as the function void baz()
union foo {
    long l;
    union baz* b; // elaborated type specifier refers to the class,
                  // not the function of the same name
};
```

Geben Sie Schlüsselwörter ein online lesen: https://riptutorial.com/de/cplusplus/topic/7838/geben-sie-schlusselworter-ein

Kapitel 50: Gewerkschaften

Bemerkungen

Gewerkschaften sind sehr nützliche Werkzeuge, haben jedoch einige wichtige Einschränkungen:

- Gemäß dem C ++ - Standard ist es undefiniertes Verhalten, auf ein Element einer Union zuzugreifen, das nicht das zuletzt geänderte Mitglied war. Obwohl viele C ++ - Compiler diesen Zugriff auf genau definierte Weise zulassen, handelt es sich dabei um Erweiterungen, die nicht für alle Compiler gewährleistet werden können.

  Eine `std::variant` (seit C ++ 17) ist wie eine Union. Sie sagt nur, was sie aktuell enthält (Teil des sichtbaren Status ist der Typ des Werts, den er zu einem bestimmten Zeitpunkt enthält: erzwingt nur den Zugriff auf Werte, der nur geschieht zu diesem Typ).

- Bei Implementierungen werden Mitglieder unterschiedlicher Größe nicht unbedingt an derselben Adresse ausgerichtet.

Examples

Grundlegende Funktionen der Union

Gewerkschaften sind eine spezialisierte Struktur, in der sich alle Mitglieder überlappendes Gedächtnis belegen.

```
union U {
    int a;
    short b;
    float c;
};
U u;

//Address of a and b will be equal
(void*)&u.a == (void*)&u.b;
(void*)&u.a == (void*)&u.c;

//Assigning to any union member changes the shared memory of all members
u.c = 4.f;
u.a = 5;
u.c != 4.f;
```

Typische Verwendung

Unions sind hilfreich, um die Speichernutzung für exklusive Daten zu minimieren, z.

```
struct AnyType {
    enum {
        IS_INT,
        IS_FLOAT
```

```
    } type;

    union Data {
        int as_int;
        float as_float;
    } value;

    AnyType(int i) : type(IS_INT) { value.as_int = i; }
    AnyType(float f) : type(IS_FLOAT) { value.as_float = f; }

    int get_int() const {
        if(type == IS_INT)
            return value.as_int;
        else
            return (int)value.as_float;
    }

    float get_float() const {
        if(type == IS_FLOAT)
            return value.as_float;
        else
            return (float)value.as_int;
    }
};
```

Undefiniertes Verhalten

```
union U {
    int a;
    short b;
    float c;
};
U u;

u.a = 10;
if (u.b == 10) {
   // this is undefined behavior since 'a' was the last member to be
   // written to. A lot of compilers will allow this and might issue a
   // warning, but the result will be "as expected"; this is a compiler
   // extension and cannot be guaranteed across compilers (i.e. this is
   // not compliant/portable code).
}
```

Gewerkschaften online lesen: https://riptutorial.com/de/cplusplus/topic/2678/gewerkschaften

Kapitel 51: Grundlegende Eingabe / Ausgabe in C ++

Bemerkungen

Die Standardbibliothek `<iostream>` definiert einige Streams für Eingabe und Ausgabe:

```
|stream | description                       |
|-------|-----------------------------------|
|cin    | standard input stream             |
|cout   | standard output stream            |
|cerr   | standard error (output) stream    |
|clog   | standard logging (output) stream |
```

Von den vier oben genannten Streams wird `cin` hauptsächlich für die Benutzereingabe verwendet, und drei andere werden für die Ausgabe der Daten verwendet. Im Allgemeinen oder in den meisten Codierungsumgebungen ist `cin` (*Konsoleneingabe* oder Standardeingabe) Tastatur und `cout` (*Konsolenausgabe* oder Standardausgabe) ist Monitor.

```
cin >> value

cin   - input stream
'>>'  - extraction operator
value - variable (destination)
```

`cin` extrahiert hier die vom Benutzer eingegebene Eingabe und gibt den variablen Wert ein. Der Wert wird erst extrahiert, wenn der Benutzer die EINGABETASTE drückt.

```
cout << "Enter a value: "

cout              - output stream
'<<'              - insertion operator
"Enter a value: " - string to be displayed
```

`cout` nimmt hier den anzuzeigenden String und fügt ihn in die Standardausgabe oder den Monitor ein

Alle vier Streams befinden sich im Standard-Namespace `std` Daher müssen Sie `std::stream` für den Stream `stream` drucken, um ihn verwenden zu können.

Es gibt auch einen Manipulator `std::endl` im Code. Es kann nur mit Ausgabeströmen verwendet werden. Es fügt das Ende der Zeile `'\n'` in den Stream ein und leert es. Dadurch wird sofort eine Ausgabe erzeugt.

Examples

Benutzereingabe und Standardausgabe

```
#include <iostream>

int main()
{
    int value;
    std::cout << "Enter a value: " << std::endl;
    std::cin >> value;
    std::cout << "The square of entered value is: " << value * value << std::endl;
    return 0;
}
```

Grundlegende Eingabe / Ausgabe in C ++ online lesen:
https://riptutorial.com/de/cplusplus/topic/10683/grundlegende-eingabe---ausgabe-in-c-plusplus

Kapitel 52: Grundtyp-Schlüsselwörter

Examples

int

Bezeichnet einen vorzeichenbehafteten Integer-Typ mit "der von der Architektur der Ausführungsumgebung vorgeschlagenen natürlichen Größe", dessen Bereich mindestens -32767 bis +32767 (einschließlich) umfasst.

```
int x = 2;
int y = 3;
int z = x + y;
```

Kann mit `unsigned` , `short` , `long` und `long long` (qv) kombiniert werden, um andere Integer-Typen zu erhalten.

bool

Ein ganzzahliger Typ, dessen Wert entweder `true` oder `false` .

```
bool is_even(int x) {
    return x%2 == 0;
}
const bool b = is_even(47); // false
```

verkohlen

Ein ganzzahliger Typ, der "groß genug ist, um ein Mitglied des Basiszeichensatzes der Implementierung zu speichern". Es ist implementierungsdefiniert, ob `char` signiert ist (und einen Bereich von mindestens -127 bis +127 (einschließlich) hat) oder unsigniert (und einen Bereich von mindestens 0 bis einschließlich 255).

```
const char zero = '0';
const char one = zero + 1;
const char newline = '\n';
std::cout << one << newline; // prints 1 followed by a newline
```

char16_t

C ++ 11

Ein vorzeichenloser Integer-Typ mit derselben Größe und Ausrichtung wie `uint_least16_t` , der daher groß genug ist, um eine UTF-16- `uint_least16_t` .

```
const char16_t message[] = u"□□□□□\n";            // Chinese for "hello, world\n"
std::cout << sizeof(message)/sizeof(char16_t) << "\n"; // prints 7
```

char32_t

C ++ 11

Ein vorzeichenloser Integer-Typ mit derselben Größe und Ausrichtung wie `uint_least32_t` , der daher groß genug ist, um eine UTF-32- `uint_least32_t` .

```
const char32_t full_house[] = U"□□□□□";               // non-BMP characters
std::cout << sizeof(full_house)/sizeof(char32_t) << "\n"; // prints 6
```

schweben

Ein Gleitkommatyp. Hat den engsten Bereich der drei Fließkommatypen in C ++.

```
float area(float radius) {
    const float pi = 3.14159f;
    return pi*radius*radius;
}
```

doppelt

Ein Gleitkommatyp. Sein Sortiment umfasst das von `float` . Wenn mit `long` kombiniert, wird der Typ des `long double` Gleitpunkts bezeichnet, dessen Bereich den Wert von `double` .

```
double area(double radius) {
    const double pi = 3.141592653589793;
    return pi*radius*radius;
}
```

lange

Bezeichnet einen vorzeichenbehafteten ganzzahligen Typ, der mindestens so lang wie `int` ist und dessen Bereich mindestens -2147483647 bis +2147483647 umfasst (d. H. - (2 ^ 31-1) bis + (2 ^ 31-1)). Dieser Typ kann auch als `long int` .

```
const long approx_seconds_per_year = 60L*60L*24L*365L;
```

Die Kombination `long double` bezeichnet einen Fließkomma-Typ, der den breitesten Bereich der drei Fließkomma-Typen aufweist.

```
long double area(long double radius) {
    const long double pi = 3.14159265358979323851;
    return pi*radius*radius;
}
```

C ++ 11

Wenn der `long` Spezifizierer zweimal auftritt, wie in `long long` , bezeichnet er einen Integer - Typ unterzeichnet, die mindestens so lang wie `long` , und deren Bereich mindestens -

9223372036854775807 bis 9223372036854775807, inclusive (das heißt, - (2 ^ 63 - 1) bis + (2 ^ 63 - 1)).

```
// support files up to 2 TiB
const long long max_file_size = 2LL << 40;
```

kurz

Bezeichnet einen vorzeichenbehafteten Integer-Typ, der mindestens so lang wie `char` ist und dessen Bereich mindestens -32767 bis +32767 (einschließlich) umfasst. Dieser Typ kann auch als `short int` .

```
// (during the last year)
short hours_worked(short days_worked) {
    return 8*days_worked;
}
```

Leere

Ein unvollständiger Typ; Es ist nicht möglich, dass ein Objekt vom Typ `void` , und es gibt keine Arrays von `void` oder Verweise auf `void` . Es wird als Rückgabetyp von Funktionen verwendet, die nichts zurückgeben.

Darüber hinaus kann eine Funktion redundant mit einem einzigen Parameter vom Typ `void` deklariert `void` . Dies ist gleichbedeutend mit der Deklaration einer Funktion ohne Parameter (zB `int main()` und `int main(void)` erklären dieselbe Funktion). Diese Syntax ist für die Kompatibilität mit C zulässig (wobei Funktionsdeklarationen eine andere Bedeutung haben als in C ++).

Der Typ `void*` ("Zeiger auf `void` ") hat die Eigenschaft, dass jeder Objektzeiger in ihn und zurück konvertiert werden kann und zum selben Zeiger führt. Diese Funktion macht den Typ `void*` für bestimmte Arten von (Typ-unsicheren) Schnittstellen zum Löschen von Typen geeignet, beispielsweise für generische Kontexte in APIs im C-Stil (z. B. `qsort` , `pthread_create`).

Jeder Ausdruck kann in einen Ausdruck vom Typ `void` . Dies wird als *Ausdruck für* einen *verworfenen Wert bezeichnet* :

```
static_cast<void>(std::printf("Hello, %s!\n", name));  // discard return value
```

Dies kann nützlich sein, um explizit zu signalisieren, dass der Wert eines Ausdrucks nicht von Interesse ist und dass der Ausdruck nur hinsichtlich seiner Nebenwirkungen bewertet werden soll.

wchar_t

Ein ganzzahliger Typ, der groß genug ist, um alle Zeichen des größten unterstützten erweiterten Zeichensatzes darzustellen, der auch als Breitzeichensatz bezeichnet wird. (Die Annahme, dass `wchar_t` eine bestimmte Kodierung wie UTF-16 verwendet, ist nicht tragbar.)

Sie wird normalerweise verwendet, wenn Sie Zeichen über ASCII 255 speichern müssen, da sie

größer als der Zeichentyp `char` .

```
const wchar_t message_ahmaric[] = L"ሰላም ልዑል\n"; //Ahmaric for "hello, world\n"
const wchar_t message_chinese[] = L"□□□□□\n";// Chinese for "hello, world\n"
const wchar_t message_hebrew[]  = L"שלום עולם\n"; //Hebrew for "hello, world\n"
const wchar_t message_russian[] = L"Привет мир\n";  //Russian for "hello, world\n"
const wchar_t message_tamil[]   = L"ஹலோ உலகம்\n"; //Tamil for "hello, world\n"
```

Grundtyp-Schlüsselwörter online lesen: https://riptutorial.com/de/cplusplus/topic/7839/grundtyp-schlusselworter

Kapitel 53: Häufige Compile / Linker-Fehler (GCC)

Examples

Fehler: '***' wurde in diesem Bereich nicht deklariert

Dieser Fehler tritt auf, wenn ein unbekanntes Objekt verwendet wird.

Variablen

Nicht kompilieren:

```
#include <iostream>

int main(int argc, char *argv[])
{
    {
        int i = 2;
    }

    std::cout << i << std::endl; // i is not in the scope of the main function

    return 0;
}
```

Fix:

```
#include <iostream>

int main(int argc, char *argv[])
{
    {
        int i = 2;
        std::cout << i << std::endl;
    }

    return 0;
}
```

Funktionen

> Meistens tritt dieser Fehler auf, wenn der erforderliche Header nicht enthalten ist (z. B. `std::cout` ohne `#include <iostream>`)

Nicht kompilieren:

```
#include <iostream>
```

```
int main(int argc, char *argv[])
{
    doCompile();

    return 0;
}

void doCompile()
{
    std::cout << "No!" << std::endl;
}
```

Fix:

```
#include <iostream>

void doCompile(); // forward declare the function

int main(int argc, char *argv[])
{
    doCompile();

    return 0;
}

void doCompile()
{
    std::cout << "No!" << std::endl;
}
```

Oder:

```
#include <iostream>

void doCompile() // define the function before using it
{
    std::cout << "No!" << std::endl;
}

int main(int argc, char *argv[])
{
    doCompile();

    return 0;
}
```

Hinweis: Der Compiler interpretiert den Code von oben nach unten (Vereinfachung). Alles muss vor der Verwendung mindestens deklariert (oder definiert) werden .

undefinierter Verweis auf "***"

Dieser Linker-Fehler tritt auf, wenn der Linker kein verwendetes Symbol findet. Meistens passiert dies, wenn eine benutzte Bibliothek nicht damit verknüpft ist.

qmake:

```
LIBS += nameOfLib
```

cmake:

```
TARGET_LINK_LIBRARIES(target nameOfLib)
```

g ++ Aufruf:

```
g++ -o main main.cpp -Llibrary/dir -lnameOfLib
```

Man könnte auch vergessen, alle verwendeten `.cpp` Dateien zu kompilieren und zu verlinken (FunctionsModule.cpp definiert die benötigte Funktion):

```
g++  -o binName main.o functionsModule.o
```

Schwerwiegender Fehler: ***: Keine solche Datei oder Verzeichnis

Der Compiler kann keine Datei finden (eine Quelldatei verwendet `#include "someFile.hpp"`).

qmake:

```
INCLUDEPATH += dir/Of/File
```

cmake:

```
include_directories(dir/Of/File)
```

g ++ Aufruf:

```
g++ -o main main.cpp -Idir/Of/File
```

Häufige Compile / Linker-Fehler (GCC) online lesen:
https://riptutorial.com/de/cplusplus/topic/4256/haufige-compile---linker-fehler--gcc-

Kapitel 54: Header-Dateien

Bemerkungen

In C ++ verwendet der C ++ - Compiler und -Kompilierungsprozess wie in C den C-Präprozessor. Wie im Handbuch zum GNU C-Präprozessor angegeben, wird eine Header-Datei wie folgt definiert:

> Eine Header-Datei ist eine Datei, die C-Deklarationen und Makrodefinitionen (siehe Makros) enthält, die von mehreren Quelldateien gemeinsam genutzt werden. Sie fordern die Verwendung einer Header-Datei in Ihrem Programm an, indem Sie sie mit der C-Vorverarbeitungs-Direktive '#include' einfügen.
>
> Header-Dateien dienen zwei Zwecken.
>
> - Systemheaderdateien deklarieren die Schnittstellen zu Teilen des Betriebssystems. Sie fügen sie in Ihr Programm ein, um die Definitionen und Deklarationen bereitzustellen, die Sie zum Aufrufen von Systemaufrufen und Bibliotheken benötigen.
> - Ihre eigenen Header-Dateien enthalten Deklarationen für Schnittstellen zwischen den Quelldateien Ihres Programms. Jedes Mal, wenn Sie eine Gruppe zusammengehöriger Deklarationen und Makrodefinitionen haben, die alle oder die meisten in mehreren verschiedenen Quelldateien benötigt werden, empfiehlt es sich, eine Headerdatei für sie zu erstellen.

Für den C-Präprozessor selbst unterscheidet sich eine Header-Datei jedoch nicht von einer Quelldatei.

Das Organisationsschema für Header- / Quellendateien ist einfach eine strenge und standardisierte Konvention, die von verschiedenen Softwareprojekten festgelegt wird, um eine Trennung zwischen Schnittstelle und Implementierung zu ermöglichen.

Obwohl dies vom C ++ - Standard selbst nicht formal durchgesetzt wird, wird die Einhaltung der Header- / Quelldatei-Konvention dringend empfohlen und ist in der Praxis bereits fast allgegenwärtig.

Beachten Sie, dass Header-Dateien als Projektdateistrukturkonvention durch die bevorstehende Funktion von Modulen ersetzt werden können, die zum Zeitpunkt des Schreibens (z. B. C ++ 20) noch in einem zukünftigen C ++ - Standard aufgenommen werden muss.

Examples

Basisbeispiel

Das folgende Beispiel enthält einen Codeblock, der in mehrere Quelldateien aufgeteilt werden soll, wie durch `// filename` .

Quelldaten

```
// my_function.h

/* Note how this header contains only a declaration of a function.
 * Header functions usually do not define implementations for declarations
 * unless code must be further processed at compile time, as in templates.
 */

/* Also, usually header files include preprocessor guards so that every header
 * is never included twice.
 *
 * The guard is implemented by checking if a header-file unique preprocessor
 * token is defined, and only including the header if it hasn't been included
 * once before.
 */
#ifndef MY_FUNCTION_H
#define MY_FUNCTION_H

// global_value and my_function() will be
// recognized as the same constructs if this header is included by different files.
const int global_value = 42;
int my_function();

#endif // MY_FUNCTION_H
```

```
// my_function.cpp

/* Note how the corresponding source file for the header includes the interface
 * defined in the header so that the compiler is aware of what the source file is
 * implementing.
 *
 * In this case, the source file requires knowledge of the global constant
 * global_value only defined in my_function.h. Without inclusion of the header
 * file, this source file would not compile.
 */
#include "my_function.h" // or #include "my_function.hpp"
int my_function() {
  return global_value; // return 42;
}
```

Header-Dateien werden dann von anderen Quelldateien eingeschlossen, die die von der Header-Schnittstelle definierte Funktionalität verwenden möchten, jedoch keine Kenntnis ihrer Implementierung erfordern (wodurch die Codekopplung reduziert wird). Das folgende Programm verwendet den Header `my_function.h` wie oben definiert:

```
// main.cpp

#include <iostream>       // A C++ Standard Library header.
#include "my_function.h"  // A personal header

int main(int argc, char** argv) {
  std::cout << my_function() << std::endl;
  return 0;
```

```
}
```

Der Kompilierungsprozess

Da Header-Dateien häufig Teil eines Kompilierungsprozess-Workflows sind, führt ein typischer Kompilierungsprozess, der die Header- / Quelldatei-Konvention verwendet, normalerweise Folgendes aus.

Wenn sich die Header- und Quellcodedatei bereits im selben Verzeichnis befinden, führt ein Programmierer die folgenden Befehle aus:

```
g++ -c my_function.cpp       # Compiles the source file my_function.cpp
                             # --> object file my_function.o

g++ main.cpp my_function.o   # Links the object file containing the
                             # implementation of int my_function()
                             # to the compiled, object version of main.cpp
                             # and then produces the final executable a.out
```

Alternativ, wenn Sie `main.cpp` in eine Objektdatei kompilieren `main.cpp` und dann als letzten Schritt nur Objektdateien miteinander verknüpfen:

```
g++ -c my_function.cpp
g++ -c main.cpp

g++ main.o my_function.o
```

Vorlagen in Header-Dateien

Vorlagen erfordern die Erzeugung von Code zur Kompilierungszeit: Eine mit Vorlagen versehene Funktion wird beispielsweise effektiv in mehrere verschiedene Funktionen umgewandelt, sobald eine mit Vorlagen versehene Funktion durch Verwendung im Quellcode parametrisiert wird.

Dies bedeutet, dass Template-Funktionen, Member-Funktionen und Klassendefinitionen nicht an eine separate Quellcodedatei delegiert werden können, da jeder Code, der ein Templat-Konstrukt verwendet, Kenntnisse seiner Definition erfordert, um generell abgeleiteten Code zu generieren.

Daher muss in der Vorlage enthaltener Code auch seine Definition enthalten, wenn er in Headern eingefügt wird. Ein Beispiel dafür ist unten:

```
// templated_function.h

template <typename T>
T* null_T_pointer() {
  T* type_point = NULL; // or, alternatively, nullptr instead of NULL
                        // for C++11 or later
  return type_point;
}
```

Header-Dateien online lesen: https://riptutorial.com/de/cplusplus/topic/7211/header-dateien

Kapitel 55: Hinterlegter Rückgabetyp

Syntax

- *Funktionsname* ([*Funktionszeichen*]) [*Funktionsattribute*] [*Funktionsqualifizierer*] `->` *Nachlauf-Rückgabetyp* [*Requires_clause*]

Bemerkungen

Die obige Syntax zeigt eine vollständige Funktionsdeklaration mit einem abschließenden Typ, wobei eckige Klammern einen optionalen Teil der Funktionsdeklaration angeben (wie die Argumentliste, wenn eine no-arg-Funktion verwendet wird).

Außerdem verhindert die Syntax des nachgestellten Rückgabetyps das Definieren eines Klassen-, Union- oder Enummentyps innerhalb eines nachlaufenden Rückgabetyps (beachten Sie, dass dies auch in einem führenden Rückgabetyp nicht zulässig ist). Ansonsten können Typen nach dem `->` die gleiche Weise buchstabiert werden wie an anderen Stellen.

Examples

Vermeiden Sie die Qualifizierung eines verschachtelten Typnamens

```
class ClassWithAReallyLongName {
  public:
    class Iterator { /* ... */ };
    Iterator end();
};
```

Definieren des Member- `end` mit einem nachgestellten Rückgabetyp:

```
auto ClassWithAReallyLongName::end() -> Iterator { return Iterator(); }
```

Definieren des Member- `end` ohne nachgestellten Rückgabetyp:

```
ClassWithAReallyLongName::Iterator ClassWithAReallyLongName::end() { return Iterator(); }
```

Der nachfolgende Rückgabetyp wird im Gültigkeitsbereich der Klasse gesucht, während ein führender Rückgabetyp im umschließenden Namespace-Bereich gesucht wird und daher eine "redundante" Qualifizierung erfordern kann.

Lambda-Ausdrücke

Ein Lambda kann *nur* eine nachlaufende Rückgabe haben. Die führende Rückgabewert-Syntax gilt nicht für Lambdas. Beachten Sie, dass es in vielen Fällen nicht erforderlich ist, für ein Lambda überhaupt einen Rückgabetyp anzugeben.

```
struct Base {};
struct Derived1 : Base {};
struct Derived2 : Base {};
auto lambda = [](bool b) -> Base* { if (b) return new Derived1; else return new Derived2; };
// ill-formed: auto lambda = Base* [](bool b) { ... };
```

Hinterlegter Rückgabetyp online lesen: https://riptutorial.com/de/cplusplus/topic/4142/hinterlegter-ruckgabetyp

Kapitel 56: Hinweise auf Mitglieder

Syntax

- Angenommen, eine Klasse namens Class ...
    - type * ptr = & Class :: member; // Nur auf statische Member zeigen
    - type Class :: * ptr = & Class :: member; // Zeigen Sie auf nicht statische Klassenmitglieder
- Für Zeiger auf nicht statische Klassenmitglieder die folgenden zwei Definitionen
    - Klasseninstanz;
    - Klasse * p = & Instanz;
- Zeiger auf Klassenmitgliedsvariablen
    - ptr = & Class :: i; // Zeige auf die Variable i innerhalb jeder Klasse
    - Instanz. * ptr = 1; // Zugriff auf die Instanz i
    - p -> * ptr = 1; // Zugriff auf p's i
- Zeiger auf Klassenmitgliedsfunktionen
    - ptr = & Class :: F; // Zeigen Sie auf die Funktion 'F' in jeder Klasse
    - (Instanz. * ptr) (5); // Aufruf der Instanz F
    - (p -> * ptr) (6); // Rufe ps F auf

Examples

Zeiger auf statische Memberfunktionen

Eine `static` Memberfunktion ist wie eine normale C / C ++ - Funktion, mit Ausnahme des Gültigkeitsbereichs:

- Es befindet sich in einer `class` und muss daher mit dem Klassennamen versehen werden.
- Es ist zugänglich, mit `public` , `protected` oder `private` .

Wenn Sie also Zugriff auf die `static` Member-Funktion haben und diese korrekt dekorieren, können Sie wie jede normale Funktion außerhalb einer `class` auf die Funktion verweisen:

```
typedef int Fn(int); // Fn is a type-of function that accepts an int and returns an int

// Note that MyFn() is of type 'Fn'
int MyFn(int i) { return 2*i; }

class Class {
public:
    // Note that Static() is of type 'Fn'
```

```
    static int Static(int i) { return 3*i; }
}; // Class

int main() {
    Fn *fn;    // fn is a pointer to a type-of Fn

    fn = &MyFn;          // Point to one function
    fn(3);               // Call it
    fn = &Class::Static; // Point to the other function
    fn(4);               // Call it
 } // main()
```

Zeiger auf Elementfunktionen

Um auf eine Member-Funktion einer Klasse zuzugreifen, benötigen Sie einen "Handle" für die jeweilige Instanz, entweder als Instanz selbst oder als Zeiger oder Verweis darauf. Bei einer Klasseninstanz können Sie auf verschiedene Member mit einem Zeiger auf Member zeigen. WENN Sie die Syntax korrekt erhalten! Natürlich muss der Zeiger so deklariert werden, dass er dem Typ entspricht, auf den Sie zeigen.

```
typedef int Fn(int); // Fn is a type-of function that accepts an int and returns an int

class Class {
public:
    // Note that A() is of type 'Fn'
    int A(int a) { return 2*a; }
    // Note that B() is of type 'Fn'
    int B(int b) { return 3*b; }
}; // Class

int main() {
    Class c;          // Need a Class instance to play with
    Class *p = &c;    // Need a Class pointer to play with

    Fn Class::*fn;    // fn is a pointer to a type-of Fn within Class

    fn = &Class::A;   // fn now points to A within any Class
    (c.*fn)(5);       // Pass 5 to c's function A (via fn)
    fn = &Class::B;   // fn now points to B within any Class
    (p->*fn)(6);      // Pass 6 to c's (via p) function B (via fn)
} // main()
```

Im Gegensatz zu Zeigern auf Member-Variablen (im vorherigen Beispiel) muss die Zuordnung zwischen der Klasseninstanz und dem Member-Zeiger eng mit Klammern verbunden sein, was ein wenig seltsam aussieht (als wären die `.*` Und `->*` nicht seltsam genug!)

Zeiger auf Membervariablen

Um auf ein Member einer `class` zuzugreifen, benötigen Sie einen "Handle" für die jeweilige Instanz, entweder als Instanz selbst oder als Zeiger oder Verweis darauf. Bei einer `class` können Sie auf verschiedene Member mit einem Zeiger auf Member zeigen. WENN Sie die Syntax korrekt erhalten! Natürlich muss der Zeiger so deklariert werden, dass er dem Typ entspricht, auf den Sie zeigen.

```
class Class {
public:
    int x, y, z;
    char m, n, o;
}; // Class

int x;  // Global variable

int main() {
    Class c;        // Need a Class instance to play with
    Class *p = &c;  // Need a Class pointer to play with

    int *p_i;       // Pointer to an int

    p_i = &x;       // Now pointing to x
    p_i = &c.x;     // Now pointing to c's x

    int Class::*p_C_i; // Pointer to an int within Class

    p_C_i = &Class::x; // Point to x within any Class
    int i = c.*p_C_i;  // Use p_c_i to fetch x from c's instance
    p_C_i = &Class::y; // Point to y within any Class
    i = c.*p_C_i;      // Use p_c_i to fetch y from c's instance

    p_C_i = &Class::m; // ERROR! m is a char, not an int!

    char Class::*p_C_c = &Class::m; // That's better...
} // main()
```

Die Syntax von pointer-to-member erfordert einige zusätzliche syntaktische Elemente:

- Um den Typ des Zeigers zu definieren, müssen Sie den Basistyp sowie die Tatsache `int Class::*ptr;` , dass er sich innerhalb einer Klasse befindet: `int Class::*ptr;` .
- Wenn Sie eine Klasse oder Referenz haben und sie benutzen wollen mit einem Zeiger-to-Mitglied, müssen Sie die verwenden `.*` Operator (verwandt mit dem `.` Operator).
- Wenn Sie einen Zeiger auf eine Klasse haben und ihn mit einem Zeiger auf Mitglied verwenden möchten, müssen Sie den Operator `->*` (ähnlich dem Operator `->`).

Zeiger auf statische Membervariablen

Eine `static` Member-Variable ist wie eine gewöhnliche C / C ++ - Variable, mit Ausnahme des Gültigkeitsbereichs:

- Es befindet sich in einer `class` und muss daher mit dem Klassennamen versehen werden.
- Es ist zugänglich, mit `public` , `protected` oder `private` .

Wenn Sie also Zugriff auf die `static` Membervariable haben und diese korrekt dekorieren, können Sie wie jede normale Variable außerhalb einer `class` auf die Variable zeigen:

```
class Class {
public:
    static int i;
}; // Class

int Class::i = 1; // Define the value of i (and where it's stored!)
```

```
int j = 2;   // Just another global variable

int main() {
    int k = 3; // Local variable

    int *p;

    p = &k;   // Point to k
    *p = 2;   // Modify it
    p = &j;   // Point to j
    *p = 3;   // Modify it
    p = &Class::i; // Point to Class::i
    *p = 4;   // Modify it
} // main()
```

Hinweise auf Mitglieder online lesen: https://riptutorial.com/de/cplusplus/topic/2130/hinweise-auf-mitglieder

Kapitel 57: Implementierungsdefiniertes Verhalten

Examples

Char ist möglicherweise nicht signiert oder signiert

Der Standard gibt nicht an, ob `char` signiert oder nicht signiert sein soll. Verschiedene Compiler implementieren sie unterschiedlich oder erlauben es möglicherweise, sie mit einem Befehlszeilenschalter zu ändern.

Größe der ganzzahligen Typen

Die folgenden Typen sind als *integrale Typen* definiert:

- `char`
- Signierte Integer-Typen
- Vorzeichenlose Integer-Typen
- `char16_t` und `char32_t`
- `bool`
- `wchar_t`

Mit Ausnahme von `sizeof(char)` / `sizeof(signed char)` / `sizeof(unsigned char)` Zeichen `sizeof(unsigned char)` , das sich in § 3.9.1.1 [basic.fundamental / 1] und § 5.3.3.1 [expr.sizeof] und `sizeof(bool)` , das vollständig implementierungsdefiniert ist und keine Mindestgröße hat, sind die Mindestgrößenanforderungen dieser Typen in Abschnitt 3.9.1 [basic.fundamental] der Norm angegeben und werden im Folgenden detailliert beschrieben.

Größe des `char`

Alle Versionen des C ++ Standard angeben, in § 5.3.3.1, dass `sizeof` Ausbeuten `1` für `unsigned char` , `signed char` und `char` (es ist festgelegt , ob die Umsetzung `char` Typ wird `signed` oder `unsigned`).

C ++ 14

`char` ist groß genug, um 256 verschiedene Werte darzustellen, um UTF-8-Codeeinheiten speichern zu können.

Größe der vorzeichenbehafteten und vorzeichenlosen Integer-Typen

Der Standard legt in § 3.9.1.2 fest, dass jeder Typ in der Liste der *signierten Integer-Typen mit Vorzeichen* , bestehend aus `signed char` , `short int` , `int` , `long int` und `long long int` , mindestens so viel Speicherplatz bereitstellen wird wie die vorhergehenden es in der Liste. Wie in § 3.9.1.3 angegeben, hat jeder dieser Typen außerdem einen entsprechenden *vorzeichenlosen Integer-Typ* mit `unsigned char` , `unsigned char` , `unsigned short int` , `unsigned int` , `unsigned long int` und ein `unsigned long long int` , das dieselbe Größe und Ausrichtung wie hat der entsprechende signierte Typ. Gemäß § 3.9.1.1 hat `char` die gleichen Größen- und Ausrichtungsanforderungen wie `signed char` und `unsigned char` .

C ++ 11

`long long` und `unsigned long long` waren vor C ++ 11 nicht offiziell Teil des C ++ - Standards. Nach der Einführung von C in C99 unterstützten viele Compiler jedoch den `long long` *Typ* als *Ganzzahl* mit `unsigned long long` und den `unsigned long long` als `unsigned long long` *Ganzzahl* mit *Vorzeichen* und den gleichen Regeln wie die C-Typen.

Der Standard garantiert somit:

```
1 == sizeof(char)  == sizeof(signed char) == sizeof(unsigned char)
  <= sizeof(short) == sizeof(unsigned short)
  <= sizeof(int)   == sizeof(unsigned int)
  <= sizeof(long)  == sizeof(unsigned long)
```

C ++ 11

```
 <= sizeof(long long) == sizeof(unsigned long long)
```

Bestimmte Mindestgrößen für jeden Typ sind in der Norm nicht angegeben. Stattdessen verfügt jeder Typ über einen minimalen Wertebereich, den er unterstützen kann. Dieser wird, wie in § 3.9.1.3 angegeben, vom C-Standard in §5.2.4.2.1 übernommen. Die Mindestgröße jedes Typs kann grob aus diesem Bereich abgeleitet werden, indem die erforderliche Mindestanzahl von Bits bestimmt wird. Beachten Sie, dass für jede Plattform der tatsächlich unterstützte Bereich eines Typs größer sein kann als das Minimum. Beachten Sie, dass die Bereiche für vorzeichenbehaftete Typen einem Komplement entsprechen, nicht dem am häufigsten verwendeten Zweierkomplement; Dies ermöglicht eine breitere Palette von Plattformen, um den Standard zu erfüllen.

Art	Mindestbereich	Minimale Bits erforderlich
`signed char`	-127 bis 127 (- (2 ^{7} - 1) bis (2 ^{7} - 1))	8
`unsigned char`	0 bis 255 (0 bis 2 ^{8} - 1)	8
`signed short`	-32,767 bis 32,767 (- (2 ^{15} - 1) bis (2 ^{15} - 1))	16
`unsigned short`	0 bis 65.535 (0 bis 2 ^{16} - 1)	16
`signed int`	-32,767 bis 32,767 (- (2	16

Art	Mindestbereich	Minimale Bits erforderlich
	^{15} - 1) bis (2 ^{15} - 1))	
`unsigned int`	0 bis 65.535 (0 bis 2 ^{16} - 1)	16
`signed long`	-2147483647 bis 2.147.483.647 (- $^{(Februar 31-01)}$ bis (31-01 Februar))	32
`unsigned long`	0 bis 4.294.967.295 (0 bis 2 ^{32} - 1)	32

C ++ 11

Art	Mindestbereich	Minimale Bits erforderlich
`signed long long`	-9.223.372.036.854.775.807 zu 9,223,372,036,854,775,807 (- (2 ^{63} - 1) bis (2 ^{63} - 1))	64
`unsigned long long`	0 bis 18.446.744.073.709.551.615 (0 bis 2 ^{64} - 1)	64

Da jeder Typ größer sein darf als seine Mindestgröße, können sich die Typen zwischen den Implementierungen unterscheiden. Das bemerkenswerteste Beispiel hierfür ist mit den 64-Bit - Datenmodellen LP64 und LLP64, wo LLP64 Systeme (wie 64-Bit - Windows) 32-Bit `int`s und `long` s und LP64 - Systeme (wie 64-Bit - Linux) haben 32-Bit - `int` s und 64-Bit `long` s. Daher kann für Integer-Typen nicht angenommen werden, dass sie auf allen Plattformen eine feste Breite haben.

C ++ 11

Wenn ganzzahlige Typen mit fester Breite erforderlich sind, verwenden Sie Typen aus dem Header `<cstdint>` Beachten Sie jedoch, dass der Standard es für Implementierungen optional macht, die Typen mit exakter Breite `int8_t` , `int16_t` , `int32_t` , `int64_t` , `intptr_t` , `uint8_t` , `uint16_t` , `uint32_t` , `uint64_t` und `uintptr_t` .

C ++ 11

Größe von `char16_t` und `char32_t`

Die Größen von `char16_t` und `char32_t` sind implementierungsdefiniert, wie in § 5.3.3.1 angegeben, mit den in § 3.9.1.5 angegebenen Bestimmungen:

- `char16_t` ist groß genug, um eine beliebige UTF-16-Codeeinheit darzustellen, und hat dieselbe Größe, Vorzeichen und Ausrichtung wie `uint_least16_t` ; es muss daher mindestens 16 Bit groß sein.
- `char32_t` ist groß genug, um eine beliebige UTF-32-Codeeinheit darzustellen, und hat dieselbe Größe, Vorzeichen und Ausrichtung wie `uint_least32_t` ; es muss daher mindestens

32 Bit groß sein.

Größe des `bool`

Die Größe von `bool` ist durch die Implementierung definiert und kann `1` sein oder nicht.

Größe von wchar_t

`wchar_t` , wie in § 3.9.1.5 angegeben, ein eindeutiger Typ, dessen Wertebereich jede unterschiedliche Codeeinheit des größten erweiterten Zeichensatzes unter den unterstützten Gebietsschemas darstellen kann. Es hat die gleiche Größe, Vorzeichen und Ausrichtung wie ein anderer integraler Typ, der als *zugrunde liegender Typ bezeichnet wird* . Die Größe dieses Typs ist implementierungsdefiniert, wie in § 5.3.3.1 angegeben, und kann beispielsweise mindestens 8, 16 oder 32 Bit betragen. Wenn ein System Unicode unterstützt, muss `wchar_t` beispielsweise mindestens 32 Bit umfassen (eine Ausnahme von dieser Regel ist Windows, wobei `wchar_t` aus Kompatibilitätsgründen 16 Bit beträgt). Es ist vom Standard C90, ISO 9899: 1990 § 4.1.5, mit nur geringfügigen Umformungen vererbt.

Je nach Implementierung beträgt die Größe von `wchar_t` oft, aber nicht immer 8, 16 oder 32 Bit. Die häufigsten Beispiele dafür sind:

- In Unix- und Unix-ähnlichen Systemen ist `wchar_t` 32-Bit und wird normalerweise für UTF-32 verwendet.
- In Windows ist `wchar_t` 16-Bit und wird für UTF-16 verwendet.
- Auf einem System, das nur 8-Bit-Unterstützung bietet, ist `wchar_t` 8 Bit.

C ++ 11

Wenn Unicode-Unterstützung gewünscht wird, wird empfohlen, `char` für UTF-8, `char16_t` für UTF-16 oder `char32_t` für UTF-32 zu verwenden, anstatt `wchar_t` .

Datenmodelle

Wie bereits erwähnt, können die Breiten von Integertypen zwischen Plattformen variieren. Die gebräuchlichsten Modelle sind folgende, wobei die Größen in Bits angegeben sind:

Modell	`int`	`long`	Zeiger
LP32 (2/4/4)	16	32	32
ILP32 (4/4/4)	32	32	32
LLP64 (4/4/8)	32	32	64

Modell	`int`	`long`	Zeiger
LP64 (4/8/8)	32	64	64

Von diesen Modellen:

- 16-Bit-Windows verwendete LP32.
- 32-Bit * Nix-Systeme (Unix, Linux, Mac OSX und andere Unix-ähnliche Betriebssysteme) und Windows verwenden ILP32.
- 64-Bit-Windows verwendet LLP64.
- 64-Bit * Nix-Systeme verwenden LP64.

Beachten Sie jedoch, dass diese Modelle im Standard selbst nicht ausdrücklich erwähnt werden.

Anzahl der Bits in einem Byte

In C ++ ist ein *Byte* der von einem `char` Objekt belegte Platz. Die Anzahl der Bits in einem Byte wird durch `CHAR_BIT` , das in `climits` definiert `climits` und mindestens 8 `climits` muss. Während die meisten modernen Systeme 8-Bit-Bytes aufweisen und für POSIX `CHAR_BIT` genau 8 ist, gibt es einige Systeme, bei denen `CHAR_BIT` ist größer als 8, dh ein einzelnes Byte kann aus 8, 16, 32 oder 64 Bits bestehen.

Numerischer Wert eines Zeigers

Das Ergebnis des Umsetzens eines Zeigers auf eine Ganzzahl mit `reinterpret_cast` ist implementierungsdefiniert, aber "... soll für diejenigen nicht überraschend sein, die die Adressierungsstruktur des zugrunde liegenden Computers kennen."

```
int x = 42;
int* p = &x;
long addr = reinterpret_cast<long>(p);
std::cout << addr << "\n"; // prints some numeric address,
                           // probably in the architecture's native address format
```

Ebenso ist der durch Konvertierung aus einer Ganzzahl erhaltene Zeiger implementierungsdefiniert.

`uintptr_t` einen Zeiger als Ganzzahl zu speichern, verwenden Sie die Typen `uintptr_t` oder `intptr_t` :

```
// `uintptr_t` was not in C++03. It's in C99, in <stdint.h>, as an optional type
#include <stdint.h>

uintptr_t uip;
```

C ++ 11

```
// There is an optional `std::uintptr_t` in C++11
#include <cstdint>
```

```
std::uintptr_t uip;
```

C ++ 11 bezieht sich auf C99 für die Definition `uintptr_t` (C99-Standard, 6.3.2.3):

> ein vorzeichenloser Integer-Typ mit der Eigenschaft, dass ein beliebiger gültiger Zeiger auf `void` in diesen Typ konvertiert und dann wieder in einen Zeiger auf `void` konvertiert `void` kann und das Ergebnis dem ursprünglichen Zeiger entspricht.

Während für die meisten modernen Plattformen ein flacher Adressraum angenommen werden kann und die Arithmetik bei `uintptr_t` der Arithmetik bei `char *` , ist es durchaus möglich, dass eine Implementierung jede Transformation ausführt, wenn `void *` in `uintptr_t` , solange die Transformation dies kann Umkehrung beim `uintptr_t` von `uintptr_t` nach `void *` .

Technische Aspekte

- Bei XSI-konformen Systemen (X / Open System Interfaces) sind die Typen `intptr_t` und `uintptr_t` erforderlich. Andernfalls sind sie **optional** .
- Funktionen im Sinne des C-Standards sind keine Objekte; Der C-Standard garantiert nicht, dass `uintptr_t` einen Funktionszeiger `uintptr_t` kann. Die Konformität von POSIX (2.12.3) erfordert jedoch Folgendes:

  > Alle Funktionszeigertypen müssen dieselbe Darstellung wie der Typzeiger auf void haben. Die Umwandlung eines Funktionszeigers in void * ändert nichts an der Darstellung. Ein void * -Wert, der sich aus einer solchen Konvertierung ergibt, kann mithilfe einer expliziten Umwandlung ohne Informationsverlust in den ursprünglichen Funktionszeigertyp konvertiert werden.

- C99 §7.18.1:

  > Wenn Typedef-Namen definiert werden, die sich nur in Abwesenheit oder Vorhandensein des anfänglichen u unterscheiden, müssen sie entsprechend vorzeichenbehaftete und vorzeichenlose Typen angeben, wie in 6.2.5 beschrieben. Eine Implementierung, die einen dieser entsprechenden Typen bereitstellt, muss auch den anderen bereitstellen.

  `uintptr_t` kann sinnvoll sein, wenn Sie Dinge mit den Bits des Zeigers machen möchten, die Sie mit einer vorzeichenbehafteten Ganzzahl nicht so sinnvoll `uintptr_t` können.

Bereiche numerischer Typen

Die Bereiche der Integer-Typen sind implementierungsdefiniert. Der Header `<limits>` enthält die Vorlage `std::numeric_limits<T>` die die Minimal- und Maximalwerte aller grundlegenden Typen `std::numeric_limits<T>` . Die Werte erfüllen die Garantien, die der C-Standard durch die `<climits>` und (> = C ++ 11) `<cinttypes>` .

- `std::numeric_limits<signed char>::min()` entspricht `SCHAR_MIN` , was kleiner oder gleich -127 ist.
- `std::numeric_limits<signed char>::max()` entspricht `SCHAR_MAX` , der größer oder gleich 127 ist.

- `std::numeric_limits<unsigned char>::max()` entspricht `UCHAR_MAX` , der größer als oder gleich 255 ist.
- `std::numeric_limits<short>::min()` entspricht `SHRT_MIN` , was kleiner oder gleich -32767 ist.
- `std::numeric_limits<short>::max()` entspricht `SHRT_MAX` , was größer oder gleich 32767 ist.
- `std::numeric_limits<unsigned short>::max()` entspricht `USHRT_MAX` , der größer als oder gleich 65535 ist.
- `std::numeric_limits<int>::min()` entspricht `INT_MIN` , was kleiner oder gleich -32767 ist.
- `std::numeric_limits<int>::max()` entspricht `INT_MAX` , der größer oder gleich 32767 ist.
- `std::numeric_limits<unsigned int>::max()` entspricht `UINT_MAX` , der größer als oder gleich 65535 ist.
- `std::numeric_limits<long>::min()` entspricht `LONG_MIN` , was gleich oder weniger als -2147483647 ist.
- `std::numeric_limits<long>::max()` entspricht `LONG_MAX` , der größer oder gleich 2147483647 ist.
- `std::numeric_limits<unsigned long>::max()` entspricht `ULONG_MAX` , der größer als oder gleich 4294967295 ist.

C ++ 11

- `std::numeric_limits<long long>::min()` entspricht `LLONG_MIN` , was kleiner oder gleich -9223372036854775807 ist.
- `std::numeric_limits<long long>::max()` entspricht `LLONG_MAX` , der größer oder gleich 9223372036854775807 ist.
- `std::numeric_limits<unsigned long long>::max()` entspricht `ULLONG_MAX` , der größer oder gleich 18446744073709551615 ist.

Für Fließkommatypen `T` ist `max()` der maximale endliche Wert, während `min()` der niedrigste positiv normalisierte Wert ist. Weitere Member werden für Gleitkommatypen bereitgestellt, die ebenfalls implementierungsdefiniert sind, jedoch bestimmte Garantien erfüllen, die der C-Standard durch den `<cfloat>` -Header `<cfloat>` .

- Das Mitglied `digits10` gibt die Anzahl der Nachkommastellen der Präzision.
  - `std::numeric_limits<float>::digits10` entspricht `FLT_DIG` , also mindestens 6.
  - `std::numeric_limits<double>::digits10` entspricht `DBL_DIG` , also mindestens 10.
  - `std::numeric_limits<long double>::digits10` entspricht `LDBL_DIG` , also mindestens 10.
- Das `min_exponent10` ist das minimale negative E, so dass 10 der Leistung E normal ist.
  - `std::numeric_limits<float>::min_exponent10` entspricht `FLT_MIN_10_EXP` , höchstens jedoch -37.
  - `std::numeric_limits<double>::min_exponent10` entspricht `DBL_MIN_10_EXP` , was höchstens -37 ist. `std::numeric_limits<long double>::min_exponent10` entspricht `LDBL_MIN_10_EXP` , höchstens jedoch -37.
- Das `max_exponent10` ist das Maximum E, so dass 10 zur Potenz E endlich ist.
  - `std::numeric_limits<float>::max_exponent10` entspricht `FLT_MIN_10_EXP` , also mindestens 37.
  - `std::numeric_limits<double>::max_exponent10` entspricht `DBL_MIN_10_EXP` , also mindestens 37.
  - `std::numeric_limits<long double>::max_exponent10` entspricht `LDBL_MIN_10_EXP` , also mindestens 37.

- Wenn der Member `is_iec559` den `is_iec559` true hat, entspricht der Typ der Norm IEC 559 / IEEE 754, und sein Bereich wird daher durch diese Norm festgelegt.

Wertdarstellung von Gleitkommatypen

Der Standard verlangt, dass `long double` mindestens so viel Präzision bietet wie `double` , was mindestens genauso viel Präzision wie `float` bietet. Ein `long double` kann einen beliebigen Wert darstellen, den ein `double` kann, während ein `double` einen beliebigen Wert darstellen kann, den ein `float` kann. Die Details der Darstellung sind jedoch implementierungsdefiniert.

Bei einem Gleitkommatyp `T` gibt `std::numeric_limits<T>::radix` die von der Darstellung von `T` verwendete `std::numeric_limits<T>::radix` .

Wenn `std::numeric_limits<T>::is_iec559` wahr ist, dann entspricht die Darstellung von `T` einem der von IEC 559 / IEEE 754 definierten Formate.

Überlauf beim Konvertieren von Ganzzahl in vorzeichenbehaftete Ganzzahl

Wenn eine vorzeichenbehaftete oder vorzeichenlose Ganzzahl in einen vorzeichenbehafteten Ganzzahlentyp konvertiert wird und ihr Wert im Zieltyp nicht darstellbar ist, wird der erzeugte Wert durch die Implementierung definiert. Beispiel:

```
// Suppose that on this implementation, the range of signed char is -128 to +127 and
// the range of unsigned char is 0 to 255
int x = 12345;
signed char sc = x;   // sc has an implementation-defined value
unsigned char uc = x; // uc is initialized to 57 (i.e., 12345 modulo 256)
```

Basiswert (und damit Größe) einer Aufzählung

Wenn der zugrunde liegende Typ nicht explizit für einen Aufzählungstyp mit nicht angegebenem Bereich angegeben ist, wird er in einer implementierungsdefinierten Weise bestimmt.

```
enum E {
    RED,
    GREEN,
    BLUE,
};
using T = std::underlying_type<E>::type; // implementation-defined
```

Der Standard erfordert jedoch, dass der zugrunde liegende Typ einer Enumeration nicht größer als `int` sei denn, `int` und `unsigned int` nicht alle Werte der Enumeration darstellen. Im obigen Code könnte `T` also `int` , `unsigned int` oder `short` , aber nicht `long long` , um einige Beispiele zu geben.

Beachten Sie, dass eine Aufzählung dieselbe Größe (wie von `sizeof`) wie ihr zugrunde liegender Typ hat.

Implementierungsdefiniertes Verhalten online lesen:
https://riptutorial.com/de/cplusplus/topic/1363/implementierungsdefiniertes-verhalten

Kapitel 58: Inline-Funktionen

Einführung

Eine mit dem `inline` Bezeichner definierte Funktion ist eine Inline-Funktion. Eine Inline-Funktion kann mehrfach definiert werden, ohne die One-Definition-Regel zu verletzen, und kann daher in einem Header mit externer Verknüpfung definiert werden. Das Deklarieren einer Funktion inline weist den Compiler an, dass die Funktion während der Codegenerierung eingebettet werden soll, bietet jedoch keine Garantie.

Syntax

- **`inline`** *function_declaration*
- **`inline`** *Funktionsdefinition*
- *Klasse {Funktionsdefinition};*

Bemerkungen

Wenn der für eine Funktion generierte Code *ausreichend* klein ist, ist er in der Regel ein guter Kandidat, um eingebettet zu werden. Warum so Wenn eine Funktion groß ist und in einer Schleife eingebettet ist, wird bei allen durchgeführten Aufrufen der Code der großen Funktion dupliziert, was zu der generierten binären Größenschwellung führt. Aber wie klein reicht aus?

Inline-Funktionen scheinen zwar eine gute Möglichkeit zu sein, um Funktionsaufrufe zu vermeiden, es ist jedoch zu beachten, dass nicht alle Funktionen, die `inline` markiert sind, `inline` sind. Mit anderen Worten, wenn Sie `inline` sagen, ist dies nur ein Hinweis an den Compiler, keine Anweisung: Der Compiler ist nicht verpflichtet, die Funktion zu integrieren, es ist frei, sie zu ignorieren - die meisten tun dies. Moderne Compiler sind besser in der Lage, solche Optimierungen vorzunehmen, dass dieses Stichwort nun ein Überbleibsel der Vergangenheit ist, als dieser Funktionsvorschlag des Programmierers von den Compilern ernst genommen wurde. Sogar Funktionen, die nicht `inline` markiert sind, werden vom Compiler eingebettet, wenn er dies für vorteilhaft hält.

Inline als Verknüpfungsrichtlinie

Der praktischere Einsatz von `inline` im modernen C ++ beruht auf der Verwendung als `inline` Direktive. Wenn Sie eine Funktion in einem Header *definieren* , jedoch nicht deklarieren, die in mehreren Quellen enthalten sein wird, hat jede Übersetzungseinheit eine eigene Kopie dieser Funktion, die zu einer ODR- Verletzung (One Definition Rule) führt. Diese Regel besagt grob, dass es nur eine Definition einer Funktion, Variablen usw. geben kann. Um diese Verletzung zu umgehen, wird die Funktionsverknüpfung implizit durch `inline` Markieren der Funktionsdefinition intern gemacht.

FAQs

Wann sollte ich das Schlüsselwort 'Inline' für eine Funktion / Methode in C ++ schreiben?

Nur wenn die Funktion in einem Header definiert werden soll. Genauer nur, wenn die Definition der Funktion in mehreren Übersetzungseinheiten angezeigt werden kann. Es ist eine gute Idee, kleine Funktionen (wie in einem Liner) in der Header-Datei zu definieren, da der Compiler damit mehr Informationen erhält, mit denen er bei der Optimierung des Codes arbeiten kann. Es erhöht auch die Kompilierungszeit.

Wann sollte ich nicht das Schlüsselwort 'Inline' für eine Funktion / Methode in C ++ schreiben?

Fügen Sie keine `inline` wenn Sie der Meinung sind, dass Ihr Code schneller ausgeführt wird, wenn der Compiler ihn `inline` einfügt.

Wann weiß der Compiler nicht, wann eine Funktion / Methode inline erstellt werden soll?

Im Allgemeinen kann der Compiler dies besser als Sie tun. Der Compiler hat jedoch keine Option zum Inline-Code, wenn er keine Funktionsdefinition hat. In maximal optimiertem Code werden normalerweise alle privaten Methoden eingebettet, ob Sie danach fragen oder nicht.

Siehe auch

- Wann sollte ich das Schlüsselwort 'Inline' für eine Funktion / Methode schreiben?
- Gibt es noch eine Verwendung für Inline?

Examples

Nicht-Mitglied-Inline-Funktionsdeklaration

```
inline int add(int x, int y);
```

Inline-Funktionsdefinition für Nichtmitglieder

```
inline int add(int x, int y)
{
    return x + y;
}
```

Member-Inline-Funktionen

```
// header (.hpp)
struct A
```

```
{
    void i_am_inlined()
    {
    }
};

struct B
{
    void i_am_NOT_inlined();
};

// source (.cpp)
void B::i_am_NOT_inlined()
{
}
```

Was ist Funktion Inlining?

```
inline int add(int x, int y)
{
    return x + y;
}

int main()
{
    int a = 1, b = 2;
    int c = add(a, b);
}
```

In dem obigen Code, wenn `add` inlined wird, würde der resultierende Code so etwas wie dieses werden

```
int main()
{
    int a = 1, b = 2;
    int c = a + b;
}
```

Die Inline-Funktion ist nirgends zu sehen, ihr Körper wird in den Körper des Anrufers *eingebettet* . Wenn `add` nicht eingebettet worden wäre, würde eine Funktion aufgerufen werden. Der Aufwand für das Aufrufen einer Funktion, z. B. das Erstellen eines neuen Stack-Frames , das Kopieren von Argumenten, das Vornehmen lokaler Variablen, das Springen (Verlust der Referenzlokalität und dort durch Cache-Misses) usw., ist erforderlich.

Inline-Funktionen online lesen: https://riptutorial.com/de/cplusplus/topic/7150/inline-funktionen

Kapitel 59: Inline-Variablen

Einführung

Eine Inline-Variable kann in mehreren Übersetzungseinheiten definiert werden, ohne dass die One-Definition-Regel verletzt wird. Wenn es mehrfach definiert ist, fügt der Linker alle Definitionen zu einem einzigen Objekt im endgültigen Programm zusammen.

Examples

Definieren eines statischen Datenelements in der Klassendefinition

Ein statisches Datenmitglied der Klasse kann in der Klassendefinition vollständig definiert sein, wenn es `inline` deklariert ist. Beispielsweise kann die folgende Klasse in einem Header definiert werden. Vor C ++ 17 musste eine `.cpp` Datei `.cpp` werden, die die Definition von `Foo::num_instances` sodass sie nur einmal definiert wird, in C ++ 17 jedoch die mehrfachen Definitionen der Inline-Variablen `Foo::num_instances` beziehen sich alle auf dasselbe `int` Objekt.

```
// warning: not thread-safe...
class Foo {
  public:
    Foo() { ++num_instances; }
    ~Foo() { --num_instances; }
    inline static int num_instances = 0;
};
```

Als Sonderfall ist ein statischer Datenbestandteil von `constexpr` implizit inline.

```
class MyString {
  public:
    MyString() { /* ... */ }
    // ...
    static constexpr int max_size = INT_MAX / 2;
};
// in C++14, this definition was required in a single translation unit:
// constexpr int MyString::max_size;
```

Inline-Variablen online lesen: https://riptutorial.com/de/cplusplus/topic/9265/inline-variablen

Kapitel 60: Intelligente Zeiger

Syntax

- `std::shared_ptr<ClassType> variableName = std::make_shared<ClassType>(arg1, arg2, ...);`
- `std::shared_ptr<ClassType> variableName (new ClassType(arg1, arg2, ...));`
- `std::unique_ptr<ClassType> variableName = std::make_unique<ClassType>(arg1, arg2, ...);` // C ++ 14
- `std::unique_ptr<ClassType> variableName (new ClassType(arg1, arg2, ...));`

Bemerkungen

C ++ ist keine speicherverwaltete Sprache. Dynamisch zugewiesener Speicher (dh Objekte, die mit `new`) wird "durchgesickert", wenn er nicht explizit freigegeben wird (mit `delete`). Es liegt in der Verantwortung des Programmierers, sicherzustellen, dass der dynamisch zugewiesene Speicher freigegeben wird, bevor der letzte Zeiger auf das Objekt gelöscht wird.

Mithilfe von intelligenten Zeigern kann der Umfang des dynamisch zugewiesenen Speichers automatisch verwaltet werden (dh, wenn der letzte Zeigerverweis seinen Gültigkeitsbereich verlässt).

Intelligente Zeiger werden in den meisten Fällen gegenüber "rohen" Zeigern bevorzugt. Sie machen die Eigentümersemantik dynamisch zugewiesenen Speichers explizit, indem sie in ihren Namen mitteilen, ob ein Objekt gemeinsam genutzt werden soll oder in eindeutigem Besitz ist.

Verwenden Sie `#include <memory>` , um intelligente Zeiger verwenden zu können.

Examples

Eigentum teilen (std :: shared_ptr)

Die Klassenvorlage `std::shared_ptr` definiert einen gemeinsamen Zeiger, der den Besitz eines Objekts mit anderen gemeinsam genutzten Zeigern teilen kann. Dies steht im Gegensatz zu `std::unique_ptr` das ausschließliches Eigentum darstellt.

Das Freigabeverhalten wird durch eine als Referenzzählung bekannte Technik implementiert, bei der die Anzahl der gemeinsam genutzten Zeiger, die auf das Objekt zeigen, daneben gespeichert wird. Wenn diese Anzahl null ist, entweder durch die Zerstörung oder Neuzuweisung der letzten Instanz von `std::shared_ptr` , wird das Objekt automatisch zerstört.

```
// Creation: 'firstShared' is a shared pointer for a new instance of 'Foo'
std::shared_ptr<Foo> firstShared = std::make_shared<Foo>(/*args*/);
```

Um mehrere intelligente Zeiger zu erstellen, die dasselbe Objekt verwenden, müssen Sie einen anderen `shared_ptr` erstellen, der den ersten gemeinsam genutzten Zeiger aliasisiert. Hier gibt es

zwei Möglichkeiten:

```
std::shared_ptr<Foo> secondShared(firstShared);  // 1st way: Copy constructing
std::shared_ptr<Foo> secondShared;
secondShared = firstShared;                      // 2nd way: Assigning
```

Mit `secondShared` eine der `secondShared` einem gemeinsamen Zeiger, der den Besitz unserer Instanz von `Foo` mit `firstShared` .

Der intelligente Zeiger funktioniert wie ein reiner Zeiger. Das heißt, Sie können sie mit `*` dereferenzieren. Der reguläre `->` Operator funktioniert ebenfalls:

```
secondShared->test(); // Calls Foo::test()
```

Wenn der letzte Aliasname `shared_ptr` Gültigkeitsbereich verlässt, wird der Destruktor unserer `Foo` Instanz aufgerufen.

Warnung: Beim `bad_alloc` eines `shared_ptr` kann eine `bad_alloc` Ausnahme `bad_alloc` wenn zusätzliche Daten für die Semantik des gemeinsam genutzten Besitzes zugewiesen werden müssen. Wenn dem Konstruktor ein regulärer Zeiger übergeben wird, wird davon ausgegangen, dass er das Objekt besitzt, auf das er zeigt, und ruft den Deleter auf, wenn eine Ausnahme ausgelöst wird. Dies bedeutet, dass `shared_ptr<T>(new T(args))` kein `T` Objekt `shared_ptr<T>` wenn die Zuweisung von `shared_ptr<T>` fehlschlägt. Es ist jedoch ratsam, `make_shared<T>(args)` oder `allocate_shared<T>(alloc, args)` , wodurch die Implementierung die Speicherzuordnung optimieren kann.

Zuweisen von Arrays ([]) mithilfe von shared_ptr

C ++ 11 C ++ 17

Leider gibt es keine direkte Möglichkeit, Arrays mithilfe von `make_shared<>` .

Es ist möglich, Arrays für `shared_ptr<>` mit `new` und `std::default_delete` .

Wenn Sie beispielsweise ein Array mit 10 Ganzzahlen zuordnen möchten, können Sie den Code als schreiben

```
shared_ptr<int> sh(new int[10], std::default_delete<int[]>());
```

Die Angabe von `std::default_delete` ist hier zwingend erforderlich, um sicherzustellen, dass der zugewiesene Speicher mithilfe von `delete[]` ordnungsgemäß bereinigt wird.

Wenn wir die Größe zur Kompilierzeit kennen, können wir dies folgendermaßen tun:

```
template<class Arr>
struct shared_array_maker {};
template<class T, std::size_t N>
struct shared_array_maker<T[N]> {
```

```
  std::shared_ptr<T> operator()const{
    auto r = std::make_shared<std::array<T,N>>();
    if (!r) return {};
    return {r.data(), r};
  }
};
template<class Arr>
auto make_shared_array()
-> decltype( shared_array_maker<Arr>{}() )
{ return shared_array_maker<Arr>{}(); }
```

dann gibt `make_shared_array<int[10]>` ein `shared_ptr<int>` das auf 10 Ints (alle standardmäßig erstellten Werte) verweist.

C ++ 17

Mit C ++ 17 erhielt `shared_ptr` spezielle Unterstützung für Array-Typen. Es ist nicht mehr erforderlich, den Array-Deleter explizit anzugeben, und der gemeinsam genutzte Zeiger kann mit dem Indexoperator `[]` dereferenziert werden:

```
std::shared_ptr<int[]> sh(new int[10]);
sh[0] = 42;
```

Gemeinsame Zeiger können auf ein Unterobjekt des Objekts zeigen, das sie besitzt:

```
struct Foo { int x; };
std::shared_ptr<Foo> p1 = std::make_shared<Foo>();
std::shared_ptr<int> p2(p1, &p1->x);
```

Sowohl `p2` als auch `p1` besitzen das Objekt vom Typ `Foo` , aber `p2` zeigt auf das `int x` . Das bedeutet, dass, wenn `p1` Gültigkeitsbereich verlässt oder neu zugewiesen wird, das zugrunde liegende `Foo` Objekt noch aktiv ist, um sicherzustellen, dass `p2` nicht baumelt.

Wichtig: Ein `shared_ptr` kennt nur sich selbst und alle anderen `shared_ptr` , die mit dem Alias-Konstruktor erstellt wurden. Es kennt keine anderen Zeiger, einschließlich aller anderen `shared_ptr` s, die mit einem Verweis auf dieselbe `Foo` Instanz erstellt wurden:

```
Foo *foo = new Foo;
std::shared_ptr<Foo> shared1(foo);
std::shared_ptr<Foo> shared2(foo); // don't do this

shared1.reset(); // this will delete foo, since shared1
                 // was the only shared_ptr that owned it

shared2->test(); // UNDEFINED BEHAVIOR: shared2's foo has been
                 // deleted already!!
```

Eigentumsübertragung von shared_ptr

In der Standardeinstellung erhöht `shared_ptr` die Referenzanzahl und überträgt den Besitz nicht. Es kann jedoch auch die Übertragung des Eigentums mit `std::move` :

```
shared_ptr<int> up = make_shared<int>();
// Transferring the ownership
shared_ptr<int> up2 = move(up);
// At this point, the reference count of up = 0 and the
// ownership of the pointer is solely with up2 with reference count = 1
```

Teilen mit temporärem Besitz (std :: weak_ptr)

Instanzen von `std::weak_ptr` können auf Objekte verweisen, deren Instanzen von `std::shared_ptr` während sie selbst nur temporäre Besitzer werden. Dies bedeutet, dass schwache Zeiger den Referenzzähler des Objekts nicht ändern und daher das Löschen eines Objekts nicht verhindern, wenn alle gemeinsam genutzten Zeiger des Objekts neu zugewiesen oder zerstört werden.

Im folgenden Beispiel werden Instanzen von `std::weak_ptr` verwendet, damit die Zerstörung eines `std::weak_ptr` nicht verhindert wird:

```
#include <memory>
#include <vector>

struct TreeNode {
    std::weak_ptr<TreeNode> parent;
    std::vector< std::shared_ptr<TreeNode> > children;
};

int main() {
    // Create a TreeNode to serve as the root/parent.
    std::shared_ptr<TreeNode> root(new TreeNode);

    // Give the parent 100 child nodes.
    for (size_t i = 0; i < 100; ++i) {
        std::shared_ptr<TreeNode> child(new TreeNode);
        root->children.push_back(child);
        child->parent = root;
    }

    // Reset the root shared pointer, destroying the root object, and
    // subsequently its child nodes.
    root.reset();
}
```

Wenn `std::weak_ptr` Knoten den `std::weak_ptr` Knoten des `std::weak_ptr` hinzugefügt werden, wird das `parent` `std::weak_ptr` auf den `std::weak_ptr` gesetzt. Das `parent` Element wird im Gegensatz zu einem gemeinsam genutzten Zeiger als schwacher Zeiger deklariert, sodass der Referenzzähler des Stammknotens nicht erhöht wird. Wenn der Wurzelknoten am Ende von `main()` , wird der Stamm zerstört. Da die einzigen noch verbliebenen `std::shared_ptr` Verweise auf die untergeordneten Knoten wurden in der Stammsammlung enthalten sind `children` , werden alle untergeordneten Knoten anschließend auch zerstört.

Aufgrund der Details der Steuerblockimplementierung kann der gemeinsam `weak_ptr` Speicher möglicherweise erst freigegeben werden, wenn der `shared_ptr` Referenzzähler und der `weak_ptr` Referenzzähler beide Null erreichen.

```
#include <memory>
int main()
{
    {
         std::weak_ptr<int> wk;
         {
             // std::make_shared is optimized by allocating only once
             // while std::shared_ptr<int>(new int(42)) allocates twice.
             // Drawback of std::make_shared is that control block is tied to our integer
             std::shared_ptr<int> sh = std::make_shared<int>(42);
             wk = sh;
             // sh memory should be released at this point...
         }
         // ... but wk is still alive and needs access to control block
     }
     // now memory is released (sh and wk)
}
```

Da `std::weak_ptr` das referenzierte Objekt nicht am Leben `std::weak_ptr` ist ein direkter Datenzugriff über ein `std::weak_ptr` nicht möglich. Stattdessen wird eine `lock() std::shared_ptr` , die versucht, ein `std::shared_ptr` für das referenzierte Objekt abzurufen:

```
#include <cassert>
#include <memory>
int main()
{
    {
         std::weak_ptr<int> wk;
         std::shared_ptr<int> sp;
         {
             std::shared_ptr<int> sh = std::make_shared<int>(42);
             wk = sh;
             // calling lock will create a shared_ptr to the object referenced by wk
             sp = wk.lock();
             // sh will be destroyed after this point, but sp is still alive
         }
         // sp still keeps the data alive.
         // At this point we could even call lock() again
         // to retrieve another shared_ptr to the same data from wk
         assert(*sp == 42);
         assert(!wk.expired());
         // resetting sp will delete the data,
         // as it is currently the last shared_ptr with ownership
         sp.reset();
         // attempting to lock wk now will return an empty shared_ptr,
         // as the data has already been deleted
         sp = wk.lock();
         assert(!sp);
         assert(wk.expired());
     }
}
```

Eindeutiger Besitz (std :: unique_ptr)

C ++ 11

Ein `std::unique_ptr` ist eine Klassenvorlage, die die Lebensdauer eines dynamisch gespeicherten

Objekts verwaltet. Im Gegensatz zu `std::shared_ptr` gehört das dynamische Objekt jederzeit nur *einer Instanz* von `std::unique_ptr` .

```
// Creates a dynamic int with value of 20 owned by a unique pointer
std::unique_ptr<int> ptr = std::make_unique<int>(20);
```

(Hinweis: `std::unique_ptr` ist seit C ++ 11 und `std::make_unique` seit C ++ 14 verfügbar.)

Nur die Variable `ptr` enthält einen Zeiger auf ein dynamisch zugewiesenes `int` . Wenn ein eindeutiger Zeiger, der ein Objekt besitzt, seinen Gültigkeitsbereich verlässt, wird das eigene Objekt gelöscht, dh der Destruktor wird aufgerufen, wenn das Objekt vom Klassentyp ist und der Speicher für dieses Objekt freigegeben wird.

Um `std::unique_ptr` und `std::make_unique` mit Array-Typen zu verwenden, verwenden Sie deren Array-Spezialisierungen:

```
// Creates a unique_ptr to an int with value 59
std::unique_ptr<int> ptr = std::make_unique<int>(59);

// Creates a unique_ptr to an array of 15 ints
std::unique_ptr<int[]> ptr = std::make_unique<int[]>(15);
```

Sie können auf den `std::unique_ptr` wie auf einen reinen Zeiger zugreifen, da diese Operatoren überladen werden.

Sie können den Besitz des Inhalts eines intelligenten Zeigers auf einen anderen Zeiger übertragen, indem Sie `std::move` , wodurch der ursprüngliche intelligente Zeiger auf `nullptr` .

```
// 1. std::unique_ptr
std::unique_ptr<int> ptr = std::make_unique<int>();

// Change value to 1
*ptr = 1;

// 2. std::unique_ptr (by moving 'ptr' to 'ptr2', 'ptr' doesn't own the object anymore)
std::unique_ptr<int> ptr2 = std::move(ptr);

int a = *ptr2; // 'a' is 1
int b = *ptr;  // undefined behavior! 'ptr' is 'nullptr'
               // (because of the move command above)
```

`unique_ptr` an Funktionen als Parameter übergeben:

```
void foo(std::unique_ptr<int> ptr)
{
    // Your code goes here
}

std::unique_ptr<int> ptr = std::make_unique<int>(59);
foo(std::move(ptr))
```

`unique_ptr` von Funktionen zurückgeben. Dies ist die bevorzugte C ++ 11-Methode zum Schreiben von Factory-Funktionen, da sie eindeutig die Besitzersemantik der Rückgabe vermittelt: Der Aufrufer besitzt das resultierende `unique_ptr` und ist dafür verantwortlich.

```
std::unique_ptr<int> foo()
{
    std::unique_ptr<int> ptr = std::make_unique<int>(59);
    return ptr;
}

std::unique_ptr<int> ptr = foo();
```

Vergleichen Sie dies mit:

```
int* foo_cpp03();

int* p = foo_cpp03(); // do I own p? do I have to delete it at some point?
                      // it's not readily apparent what the answer is.
```

C ++ 14

Die Klassenvorlage `make_unique` wird seit C ++ 14 bereitgestellt. Es ist einfach, es manuell zu C ++ 11-Code hinzuzufügen:

```
template<typename T, typename... Args>
typename std::enable_if<!std::is_array<T>::value, std::unique_ptr<T>>::type
make_unique(Args&&... args)
{ return std::unique_ptr<T>(new T(std::forward<Args>(args)...)); }

// Use make_unique for arrays
template<typename T>
typename std::enable_if<std::is_array<T>::value, std::unique_ptr<T>>::type
make_unique(size_t n)
{ return std::unique_ptr<T>(new typename std::remove_extent<T>::type[n]()); }
```

C ++ 11

Im Gegensatz zum *dummen* intelligenten Zeiger (`std::auto_ptr`) kann `unique_ptr` auch mit Vektorzuordnung (*nicht* `std::vector`) instanziiert werden. Frühere Beispiele waren für *Skalarzuweisungen* . Wenn Sie beispielsweise ein dynamisch zugewiesenes Integer-Array für 10 Elemente haben möchten, geben Sie als Vorlagentyp `int[]` (und nicht nur als `int`):

```
std::unique_ptr<int[]> arr_ptr = std::make_unique<int[]>(10);
```

Was kann vereinfacht werden mit:

```
auto arr_ptr = std::make_unique<int[]>(10);
```

Jetzt verwenden Sie `arr_ptr` als wäre es ein Array:

```
arr_ptr[2] =  10; // Modify third element
```

Sie müssen sich keine Gedanken über die Aufteilung machen. Diese vorlagenspezifische Version ruft Konstruktoren und Destruktoren entsprechend auf. Die Verwendung einer vektorisierten Version von `unique_ptr` oder eines `vector` selbst ist eine persönliche Entscheidung.

In Versionen vor C ++ 11 war `std::auto_ptr` verfügbar. Im Gegensatz zu `unique_ptr` es erlaubt, `auto_ptr` s zu kopieren, wobei der Quell- `ptr` den Besitz des enthaltenen Zeigers verliert und das Ziel ihn erhält.

Verwenden von benutzerdefinierten Deleters, um einen Wrapper für eine C-Schnittstelle zu erstellen

Viele C-Schnittstellen wie SDL2 verfügen über eigene Löschfunktionen . Das bedeutet, dass Sie intelligente Zeiger nicht direkt verwenden können:

```
std::unique_ptr<SDL_Surface> a; // won't work, UNSAFE!
```

Stattdessen müssen Sie Ihren eigenen Deleter definieren. Die Beispiele hier verwenden die `SDL_Surface` Struktur, die mit der Funktion `SDL_FreeSurface()` freigegeben werden sollte, sie sollten jedoch an viele andere C-Schnittstellen `SDL_FreeSurface()` werden können.

Der Deleter muss mit einem Zeigerargument aufrufbar sein und kann daher zB ein einfacher Funktionszeiger sein:

```
std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)> a(pointer, SDL_FreeSurface);
```

Jedes andere aufrufbare Objekt funktioniert ebenfalls, beispielsweise eine Klasse mit einem `operator()` :

```
struct SurfaceDeleter {
    void operator()(SDL_Surface* surf) {
        SDL_FreeSurface(surf);
    }
};

std::unique_ptr<SDL_Surface, SurfaceDeleter> a(pointer, SurfaceDeleter{}); // safe
std::unique_ptr<SDL_Surface, SurfaceDeleter> b(pointer); // equivalent to the above
                                                         // as the deleter is value-
initialized
```

Dadurch erhalten Sie nicht nur eine automatische, automatische Speicherverwaltung, die keinen Overhead erfordert (wenn Sie `unique_ptr`), sondern auch die Sicherheit von Ausnahmen.

Beachten Sie, dass der Deleter Teil des Typs für `unique_ptr` ist und die Implementierung die leere Basisoptimierung verwenden kann, um jegliche Größenänderung für leere benutzerdefinierte Deleter zu vermeiden. Während also `std::unique_ptr<SDL_Surface, SurfaceDeleter>` und `std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)>` dasselbe Problem auf ähnliche Weise lösen, ist der vorherige Typ immer nur die Größe eines Zeigers letzterer Typ muss *zwei* Zeiger enthalten: sowohl `SDL_Surface*` als auch den Funktionszeiger! Wenn Sie benutzerdefinierte Deleter für freie Funktionen verwenden, ist es vorzuziehen, die Funktion in einem leeren Typ einzugeben.

In Fällen, in denen die Referenzzählung wichtig ist, kann anstelle von `unique_ptr` ein `shared_ptr` verwendet werden. Der `shared_ptr` speichert immer einen Deleter, wodurch der Typ des Deleters gelöscht wird, was in APIs nützlich sein kann. Die Nachteile der Verwendung von `shared_ptr` gegenüber `unique_ptr` sind höhere Speicherkosten für die Speicherung des Deleters und Leistungskosten für die Aufrechterhaltung der Referenzzählung.

```
// deleter required at construction time and is part of the type
std::unique_ptr<SDL_Surface, void(*)(SDL_Surface*)> a(pointer, SDL_FreeSurface);

// deleter is only required at construction time, not part of the type
std::shared_ptr<SDL_Surface> b(pointer, SDL_FreeSurface);
```

C ++ 17

Mit `template auto` können wir unsere benutzerdefinierten Deleter noch einfacher verpacken:

```
template <auto DeleteFn>
struct FunctionDeleter {
    template <class T>
    void operator()(T* ptr) {
        DeleteFn(ptr);
    }
};

template <class T, auto DeleteFn>
using unique_ptr_deleter = std::unique_ptr<T, FunctionDeleter<DeleteFn>>;
```

Bei dem das obige Beispiel ist einfach:

```
unique_ptr_deleter<SDL_Surface, SDL_FreeSurface> c(pointer);
```

Der Zweck von `auto` ist es, alle freien Funktionen zu behandeln, unabhängig davon, ob sie `void` (z. B. `SDL_FreeSurface`) oder nicht (z. B. `fclose`).

Einzigartiger Besitz ohne Umzugssemantik (auto_ptr)

C ++ 11

Hinweis: `std::auto_ptr` wurde in C ++ 11 nicht mehr unterstützt und wird in C ++ 17 entfernt. Sie sollten dies nur verwenden, wenn Sie C ++ 03 oder eine frühere Version verwenden müssen und vorsichtig sein möchten. Es wird empfohlen, zu unique_ptr in Kombination mit `std::move` zu `std::move` , um `std::auto_ptr` Verhalten von `std::auto_ptr` zu ersetzen.

Bevor wir `std::unique_ptr` , bevor wir die Semantik des Verschiebens hatten, hatten wir `std::auto_ptr` . `std::auto_ptr` bietet einen eindeutigen Besitz, überträgt jedoch den Besitz beim Kopieren.

Wie bei allen intelligenten Zeigern bereinigt `std::auto_ptr` automatisch Ressourcen (siehe RAII):

```
{
    std::auto_ptr<int> p(new int(42));
```

```
    std::cout << *p;
} // p is deleted here, no memory leaked
```

erlaubt aber nur einen Besitzer:

```
std::auto_ptr<X> px = ...;
std::auto_ptr<X> py = px;
  // px is now empty
```

Dies ermöglicht die Verwendung von std :: auto_ptr, um den Besitz explizit und eindeutig zu halten, da die Gefahr besteht, dass der Besitz unbeabsichtigt verloren geht:

```
void f(std::auto_ptr<X> ) {
    // assumes ownership of X
    // deletes it at end of scope
};

std::auto_ptr<X> px = ...;
f(px); // f acquires ownership of underlying X
       // px is now empty
px->foo(); // NPE!
// px.~auto_ptr() does NOT delete
```

Die Eigentumsübertragung erfolgte im Konstruktor "copy". `auto_ptr` und der `auto_ptr` ihre Operanden anhand einer nicht `const` Referenz, sodass sie geändert werden können. Eine Beispielimplementierung könnte sein:

```
template <typename T>
class auto_ptr {
    T* ptr;
public:
    auto_ptr(auto_ptr& rhs)
    : ptr(rhs.release())
    { }

    auto_ptr& operator=(auto_ptr& rhs) {
        reset(rhs.release());
        return *this;
    }

    T* release() {
        T* tmp = ptr;
        ptr = nullptr;
        return tmp;
    }

    void reset(T* tmp = nullptr) {
        if (ptr != tmp) {
            delete ptr;
            ptr = tmp;
        }
    }

    /* other functions ... */
};
```

Dies unterbricht die Kopiersemantik, bei der Sie beim Kopieren eines Objekts zwei gleichwertige Versionen hinterlassen müssen. Für jeden kopierbaren Typ `T` sollte ich schreiben können:

```
T a = ...;
T b(a);
assert(b == a);
```

Bei `auto_ptr` ist dies jedoch nicht der Fall. Daher ist es nicht sicher, `auto_ptr` s in Containern `auto_ptr` .

Get ein shared_ptr, der sich darauf bezieht

`enable_shared_from_this` können Sie eine gültige `shared_ptr` Instanz auf `this` `enable_shared_from_this` .

Indem Sie Ihre Klasse von der Klassenvorlage `enable_shared_from_this` , erben Sie eine Methode `shared_from_this` , die eine `shared_ptr` Instanz an `this` zurückgibt.

Beachten Sie, dass das Objekt an erster Stelle als `shared_ptr` erstellt werden muss:

```
#include <memory>
class A: public enable_shared_from_this<A> {
};
A* ap1 =new A();
shared_ptr<A> ap2(ap1); // First prepare a shared pointer to the object and hold it!
// Then get a shared pointer to the object from the object itself
shared_ptr<A> ap3 = ap1->shared_from_this();
int c3 =ap3.use_count(); // =2: pointing to the same object
```

Hinweis (2) Sie können `enable_shared_from_this` innerhalb des Konstruktors aufrufen.

```
#include <memory> // enable_shared_from_this

class Widget : public std::enable_shared_from_this< Widget >
{
public:
    void DoSomething()
    {
        std::shared_ptr< Widget > self = shared_from_this();
        someEvent -> Register( self );
    }
private:
    ...
};

int main()
{
    ...
    auto w = std::make_shared< Widget >();
    w -> DoSomething();
    ...
}
```

Wenn Sie `shared_from_this()` für ein Objekt verwenden, das nicht zu einem `shared_ptr` , z. B. ein

lokales automatisches Objekt oder ein globales Objekt, ist das Verhalten undefiniert. Seit C ++ 17 wird stattdessen `std::bad_alloc` .

Die Verwendung von `shared_from_this()` von einem Konstruktor ist gleichbedeutend mit der Verwendung für ein Objekt, das sich nicht im Besitz eines `shared_ptr` , da die Objekte nach der Rückgabe des Konstruktors im Besitz des `shared_ptr` .

Casting von std :: shared_ptr-Zeigern

Es ist nicht möglich, `static_cast` , `const_cast` , `dynamic_cast` und `reinterpret_cast` direkt für `std::shared_ptr` zu verwenden, um einen Zeiger abzurufen, der den Besitz des als Argument übergebenen Zeigers teilt. Stattdessen sollten die Funktionen `std::static_pointer_cast` , `std::const_pointer_cast` , `std::dynamic_pointer_cast` und `std::reinterpret_pointer_cast` verwendet werden:

```
struct Base { virtual ~Base() noexcept {}; };
struct Derived: Base {};
auto derivedPtr(std::make_shared<Derived>());
auto basePtr(std::static_pointer_cast<Base>(derivedPtr));
auto constBasePtr(std::const_pointer_cast<Base const>(basePtr));
auto constDerivedPtr(std::dynamic_pointer_cast<Derived const>(constBasePtr));
```

Beachten Sie, dass `std::reinterpret_pointer_cast` in C ++ 11 und C ++ 14 nicht verfügbar ist, da es nur von N3920 vorgeschlagen und im Februar 2014 in Library Fundamentals TS übernommen wurde . Es kann jedoch wie folgt implementiert werden:

```
template <typename To, typename From>
inline std::shared_ptr<To> reinterpret_pointer_cast(
    std::shared_ptr<From> const & ptr) noexcept
{ return std::shared_ptr<To>(ptr, reinterpret_cast<To *>(ptr.get())); }
```

Einen intelligenten Zeiger schreiben: value_ptr

Ein `value_ptr` ist ein intelligenter Zeiger, der sich wie ein Wert verhält. Beim Kopieren wird der Inhalt kopiert. Wenn erstellt, erstellt es seinen Inhalt.

```
// Like std::default_delete:
template<class T>
struct default_copier {
  // a copier must handle a null T const* in and return null:
  T* operator()(T const* tin)const {
    if (!tin) return nullptr;
    return new T(*tin);
  }
  void operator()(void* dest, T const* tin)const {
    if (!tin) return;
    return new(dest) T(*tin);
  }
};
// tag class to handle empty case:
struct empty_ptr_t {};
constexpr empty_ptr_t empty_ptr{};
```

```
// the value pointer type itself:
template<class T, class Copier=default_copier<T>, class Deleter=std::default_delete<T>,
  class Base=std::unique_ptr<T, Deleter>
>
struct value_ptr:Base, private Copier {
  using copier_type=Copier;
  // also typedefs from unique_ptr

  using Base::Base;

  value_ptr( T const& t ):
    Base( std::make_unique<T>(t) ),
    Copier()
  {}
  value_ptr( T && t ):
    Base( std::make_unique<T>(std::move(t)) ),
    Copier()
  {}
  // almost-never-empty:
     value_ptr():
    Base( std::make_unique<T>() ),
    Copier()
  {}
  value_ptr( empty_ptr_t ) {}

  value_ptr( Base b, Copier c={} ):
    Base(std::move(b)),
    Copier(std::move(c))
  {}

  Copier const& get_copier() const {
    return *this;
  }

  value_ptr clone() const {
    return {
      Base(
        get_copier()(this->get()),
        this->get_deleter()
      ),
      get_copier()
    };
  }
  value_ptr(value_ptr&&)=default;
  value_ptr& operator=(value_ptr&&)=default;

  value_ptr(value_ptr const& o):value_ptr(o.clone()) {}
  value_ptr& operator=(value_ptr const&o) {
    if (o && *this) {
      // if we are both non-null, assign contents:
      **this = *o;
    } else {
      // otherwise, assign a clone (which could itself be null):
      *this = o.clone();
    }
    return *this;
  }
  value_ptr& operator=( T const& t ) {
    if (*this) {
      **this = t;
    } else {
```

```
      *this = value_ptr(t);
    }
    return *this;
  }
  value_ptr& operator=( T && t ) {
    if (*this) {
      **this = std::move(t);
    } else {
      *this = value_ptr(std::move(t));
    }
    return *this;
  }
  T& get() { return **this; }
  T const& get() const { return **this; }
  T* get_pointer() {
    if (!*this) return nullptr;
    return std::addressof(get());
  }
  T const* get_pointer() const {
    if (!*this) return nullptr;
    return std::addressof(get());
  }
  // operator-> from unique_ptr
};
template<class T, class...Args>
value_ptr<T> make_value_ptr( Args&&... args ) {
  return {std::make_unique<T>(std::forward<Args>(args)...)};
}
```

Dieses bestimmte value_ptr ist nur leer, wenn Sie es mit `empty_ptr_t` oder wenn Sie es verlassen. Es macht die Tatsache `unique_ptr` , dass es sich um ein `unique_ptr` , so dass der `explicit operator bool() const` funktioniert. `.get()` wurde geändert, um einen Verweis zurückzugeben (da er fast nie leer ist), und `.get_pointer()` gibt stattdessen einen Zeiger zurück.

Dieser intelligente Zeiger kann in `pImpl` Fällen nützlich `pImpl` , in denen `pImpl` wird, der Inhalt von `pImpl` außerhalb der Implementierungsdatei `pImpl` werden soll.

Mit einem nicht standardmäßigen `Copier` kann er sogar virtuelle Basisklassen verarbeiten, die wissen, wie Instanzen ihrer abgeleiteten Objekte erzeugt und in Werttypen umgewandelt werden.

Intelligente Zeiger online lesen: https://riptutorial.com/de/cplusplus/topic/509/intelligente-zeiger

Kapitel 61: Internationalisierung in C ++

Bemerkungen

Die Sprache C ++ diktiert keinen Zeichensatz, einige Compiler die Verwendung von UTF-8 ***unterstützen*** können, oder auch UTF-16. Es besteht jedoch keine Gewissheit, dass etwas anderes als einfache ANSI / ASCII-Zeichen bereitgestellt wird.

Daher ist die gesamte internationale Sprachunterstützung durch die Implementierung definiert, abhängig davon, welche Plattform, welches Betriebssystem und welcher Compiler Sie verwenden.

Mehrere Drittanbieter-Bibliotheken (wie die International Unicode Committee Library), mit denen die internationale Unterstützung der Plattform erweitert werden kann.

Examples

C ++ - Zeichenfolgenmerkmale verstehen

```
#include <iostream>
#include <string>

int main()
{
    const char * C_String = "This is a line of text w";
    const char * C_Problem_String = "This is a line of text ";
  std::string Std_String("This is a second line of text w");
    std::string Std_Problem_String("This is a second line of ex ");

  std::cout << "String Length: " << Std_String.length() << '\n';
    std::cout << "String Length: " << Std_Problem_String.length() << '\n';

    std::cout << "CString Length: " << strlen(C_String) << '\n';
    std::cout << "CString Length: " << strlen(C_Problem_String) << '\n';
    return 0;
}
```

Je nach Plattform (Windows, OSX usw.) und Compiler (GCC, MSVC usw.) wird dieses Programm **möglicherweise nicht kompiliert, zeigt andere Werte an oder zeigt dieselben Werte an** .

Beispielausgabe unter dem Microsoft MSVC-Compiler:

> Saitenlänge: 31
> Saitenlänge: 31
> CS-Länge: 24
> CS-Länge: 24

Dies zeigt, dass unter MSVC jedes der verwendeten erweiterten Zeichen als einzelnes "Zeichen" betrachtet wird und diese Plattform international unterstützte Sprachen vollständig unterstützt.
Es ist jedoch zu beachten, dass dieses Verhalten ungewöhnlich ist. Diese internationalen Zeichen

werden intern als Unicode gespeichert und sind daher tatsächlich mehrere Bytes lang. ***Dies kann zu unerwarteten Fehlern führen***

Unter dem GNC / GCC-Compiler lautet die Programmausgabe:

> Saitenlänge: 31
> Saitenlänge: 36
> CS-Länge: 24
> CS-Länge: 26

Dieses Beispiel zeigt, dass der auf dieser (Linux-) Plattform verwendete GCC-Compiler zwar diese erweiterten Zeichen unterstützt, dass er jedoch mehrere Bytes (*richtig) verwendet* , um ein einzelnes Zeichen zu speichern.
In diesem Fall ist die Verwendung von Unicode-Zeichen möglich, der Programmierer muss jedoch sorgfältig darauf achten, dass die Länge einer "Zeichenfolge" in diesem Szenario der ***Länge in Byte*** und nicht der ***Länge in lesbaren Zeichen entspricht*** .

Diese Unterschiede sind darauf zurückzuführen, wie internationale Sprachen auf Plattformbasis gehandhabt werden - und was noch wichtiger ist, dass die in diesem Beispiel verwendeten C- und C ++ - Zeichenfolgen als ***ein Array von Bytes betrachtet werden können*** , sodass (für diese Verwendung) ***die C ++ - Sprache berücksichtigt wird ein Zeichen (Zeichen), um ein einzelnes Byte zu sein*** .

Internationalisierung in C ++ online lesen:
https://riptutorial.com/de/cplusplus/topic/5270/internationalisierung-in-c-plusplus

Kapitel 62: Iteration

Examples

brechen

Springt aus der nächstgelegenen Umschließungsschleife oder `switch` .

```
// print the numbers to a file, one per line
for (const int num : num_list) {
    errno = 0;
    fprintf(file, "%d\n", num);
    if (errno == ENOSPC) {
        fprintf(stderr, "no space left on device; output will be truncated\n");
        break;
    }
}
```

fortsetzen

Springt zum Ende der kleinsten Umschließungsschleife.

```
int sum = 0;
for (int i = 0; i < N; i++) {
    int x;
    std::cin >> x;
    if (x < 0) continue;
    sum += x;
    // equivalent to: if (x >= 0) sum += x;
}
```

tun

Leitet eine Do-while-Schleife ein .

```
// Gets the next non-whitespace character from standard input
char read_char() {
    char c;
    do {
        c = getchar();
    } while (isspace(c));
    return c;
}
```

zum

Führt eine for-Schleife oder in C ++ 11 und höher eine bereichsbasierte for-Schleife ein .

```
// print 10 asterisks
for (int i = 0; i < 10; i++) {
```

```
    putchar('*');
}
```

während

Leitet eine while-Schleife ein .

```
int i = 0;
// print 10 asterisks
while (i < 10) {
    putchar('*');
    i++;
}
```

bereichsbasiert für Schleife

```
std::vector<int> primes = {2, 3, 5, 7, 11, 13};

for(auto prime : primes) {
    std::cout << prime << std::endl;
}
```

Iteration online lesen: https://riptutorial.com/de/cplusplus/topic/7841/iteration

Kapitel 63: Iteratoren

Examples

C Iteratoren (Zeiger)

```
// This creates an array with 5 values.
const int array[] = { 1, 2, 3, 4, 5 };

#ifdef BEFORE_CPP11

// You can use `sizeof` to determine how many elements are in an array.
const int* first = array;
const int* afterLast = first + sizeof(array) / sizeof(array[0]);

// Then you can iterate over the array by incrementing a pointer until
// it reaches past the end of our array.
for (const int* i = first; i < afterLast; ++i) {
    std::cout << *i << std::endl;
}

#else

// With C++11, you can let the STL compute the start and end iterators:
for (auto i = std::begin(array); i != std::end(array); ++i) {
    std::cout << *i << std::endl;
}

#endif
```

Dieser Code würde die Zahlen 1 bis 5 ausgeben, eine in jeder Zeile wie folgt:

1
2
3
4
5

Brechen sie ab

```
const int array[] = { 1, 2, 3, 4, 5 };
```

Diese Zeile erstellt ein neues Integer-Array mit 5 Werten. C-Arrays sind nur Zeiger auf Speicher, in denen jeder Wert zusammen in einem zusammenhängenden Block gespeichert wird.

```
const int* first = array;
const int* afterLast = first + sizeof(array) / sizeof(array[0]);
```

Diese Linien erzeugen zwei Zeiger. Der erste Zeiger erhält den Wert des Arrayzeigers, der die Adresse des ersten Elements im Array ist. Der Operator `sizeof` bei Verwendung in einem C-Array

die Größe des Arrays in Byte zurück. Geteilt durch die Größe eines Elements ergibt sich die Anzahl der Elemente im Array. Damit können wir die Adresse des Blocks *nach* dem Array ermitteln.

```
for (const int* i = first; i < afterLast; ++i) {
```

Hier erstellen wir einen Zeiger, den wir als Iterator verwenden werden. Es wird mit der Adresse des ersten Elements wir iterieren wollen initialisiert und wir werden so lange iteriert weiterhin als `i` kleiner als ist `afterLast` , die so lange Mittel wie `i` innerhalb einer Adresse zeigt `array` .

```
    std::cout << *i << std::endl;
```

Schließlich innerhalb der Schleife können wir den Wert unserer Iterator zugreifen `i` zu zeigt , indem sie es dereferencing. Hier gibt der Dereferenzierungsoperator `*` den Wert an der Adresse in `i` .

Überblick

Iteratoren sind Positionen

Iteratoren sind ein Mittel zum Navigieren und Bedienen einer Folge von Elementen und sind eine verallgemeinerte Erweiterung von Zeigern. Konzeptionell ist es wichtig zu wissen, dass Iteratoren Positionen und keine Elemente sind. Nehmen Sie zum Beispiel die folgende Reihenfolge:

```
A B C
```

Die Sequenz enthält drei Elemente und vier Positionen

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
```

Elemente sind Dinge innerhalb einer Sequenz. Positionen sind Orte, an denen sinnvolle Operationen mit der Sequenz passieren können. Zum Beispiel fügt man in eine Position *vor* oder *nach* Element A ein, nicht in ein Element. Selbst das Löschen eines Elements (`erase(A)`) erfolgt, indem zuerst die Position ermittelt und anschließend gelöscht wird.

Von Iteratoren zu Werten

Um von einer Position in einen Wert umzuwandeln, wird ein Iterator *dereferenziert* :

```
auto my_iterator = my_vector.begin(); // position
auto my_value = *my_iterator; // value
```

Man kann sich einen Iterator als Dereferenzierung auf den Wert vorstellen, auf den er in der

Sequenz verweist. Dies ist besonders nützlich, um zu verstehen, warum Sie niemals den `end()` Iterator in einer Sequenz dereferenzieren sollten:

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
  ↑           ↑
  |           +-- An iterator here has no value. Do not dereference it!
  +-------------- An iterator here dereferences to the value A.
```

In allen Sequenzen und Containern, die in der C ++ - Standardbibliothek zu finden sind, bringt `begin()` einen Iterator an die erste Position zurück und `end()` einen Iterator an die letzte Position (*nicht an* die letzte Position!). Daher werden die Namen dieser Iteratoren in Algorithmen oft als `first` und `last` :

```
+---+---+---+---+
| A | B | C |   |
+---+---+---+---+
  ↑           ↑
  |           |
  +- first    +- last
```

Es ist auch möglich, einen Iterator für eine *beliebige Sequenz zu erhalten* , da selbst eine leere Sequenz mindestens eine Position enthält:

```
+---+
|   |
+---+
```

In einer leeren Sequenz sind `begin()` und `end()` die gleiche Position und *keine* dereferenziert werden kann:

```
+---+
|   |
+---+
  ↑
  |
  +- empty_sequence.begin()
  |
  +- empty_sequence.end()
```

Die alternative Visualisierung von Iteratoren besteht darin, dass sie die Positionen *zwischen den* Elementen markieren:

```
+---+---+---+
| A | B | C |
+---+---+---+
↑   ^   ^   ↑
|           |
+- first    +- last
```

und dereferenzieren eines Iterators gibt eine Referenz auf das Element zurück, das nach dem

Iterator kommt. Einige Situationen, in denen diese Ansicht besonders nützlich ist, sind:

- `insert` werden Elemente in die vom Iterator angegebene Position eingefügt.
- `erase` geben einen Iterator zurück, der der gleichen Position entspricht wie der übergebene,
- Ein Iterator und sein entsprechender Reverse-Iterator befinden sich in derselben Position zwischen den Elementen

Ungültige Iteratoren

Ein Iterator wird *ungültig,* wenn (z. B. während einer Operation) seine Position nicht mehr Teil einer Sequenz ist. Ein ungültiger Iterator kann nicht dereferenziert werden, bis er einer gültigen Position zugewiesen wurde. Zum Beispiel:

```
std::vector<int>::iterator first;
{
    std::vector<int> foo;
    first = foo.begin(); // first is now valid
} // foo falls out of scope and is destroyed
// At this point first is now invalid
```

Für die vielen Algorithmen und Sequenzelementfunktionen in der C ++ - Standardbibliothek gelten Regeln, wenn Iteratoren ungültig gemacht werden. Jeder Algorithmus unterscheidet sich in der Art und Weise, wie er Iteratoren behandelt (und ungültig macht).

Mit Iteratoren navigieren

Wie wir wissen, sind Iteratoren für das Navigieren von Sequenzen. Zu diesem Zweck muss ein Iterator seine Position während der gesamten Sequenz verschieben. Iteratoren können in der Sequenz vorwärts und einige rückwärts vorrücken:

```
auto first = my_vector.begin();
++first;                                            // advance the iterator 1 position
std::advance(first, 1);                             // advance the iterator 1 position
first = std::next(first);                           // returns iterator to the next element
std::advance(first, -1);                            // advance the iterator 1 position
backwards
first = std::next(first, 20);                       // returns iterator to the element 20
position forward
first = std::prev(first, 5);                        // returns iterator to the element 5
position backward
auto dist = std::distance(my_vector.begin(), first); // returns distance between two
iterators.
```

Beachten Sie, dass das zweite Argument von std :: distance vom ersten Argument aus erreichbar sein sollte (oder anders ausgedrückt, das `first` sollte kleiner oder gleich dem `second`).

Obwohl Sie arithmetische Operatoren mit Iteratoren ausführen können, sind nicht alle Operationen für alle Arten von Iteratoren definiert. `a = b + 3;` würde für Random Access Iterators funktionieren, aber nicht für Forward- oder Bidirectional Iterators, die immer noch um 3 Positionen mit etwas wie

`b = a; ++b; ++b; ++b;` vorgerückt werden können `b = a; ++b; ++b; ++b;` . Es wird daher empfohlen, spezielle Funktionen zu verwenden, wenn Sie nicht sicher sind, um welchen Iteratortyp es sich handelt (z. B. in einer Vorlagenfunktion, die den Iterator akzeptiert).

Iterator-Konzepte

Der C ++ - Standard beschreibt verschiedene Iteratorkonzepte. Diese werden nach ihrem Verhalten in den Sequenzen gruppiert, auf die sie sich beziehen. Wenn Sie das Konzept kennen, das ein Iterator *modelliert* (verhält sich wie), können Sie sich auf das Verhalten dieses Iterators verlassen, *unabhängig von der Sequenz, zu der er gehört* . Sie werden oft in der Reihenfolge von am wenigsten bis am wenigsten einschränkend beschrieben (da das nächste Iteratorkonzept einen Schritt besser ist als sein Vorgänger):

- Eingabe-Iteratoren: Kann *nur einmal* pro Position dereferenziert werden. Kann nur vorrücken und jeweils nur eine Position.
- Forward-Iteratoren: Ein Eingabe-Iterator, der beliebig oft dereferenziert werden kann.
- Bidirektionale Iteratoren: Ein Vorwärts-Iterator, der auch eine Position nach *hinten* vorrücken kann.
- Iteratoren mit wahlfreiem Zugriff: Ein bidirektionaler Iterator, der eine beliebige Anzahl von Positionen gleichzeitig vor- oder zurückbewegen kann.
- Angrenzende Iteratoren (seit C ++ 17): Ein Iterator mit wahlfreiem Zugriff, der garantiert, dass die zugrunde liegenden Daten im Speicher zusammenhängend sind.

Algorithmen können je nach Konzept variieren, das von den angegebenen Iteratoren modelliert wird. Obwohl `random_shuffle` beispielsweise für Forward-Iteratoren `random_shuffle` kann, könnte eine effizientere Variante bereitgestellt werden, die Iteratoren mit wahlfreiem Zugriff erfordert.

Iterator-Merkmale

Iterator-Merkmale bieten eine einheitliche Schnittstelle zu den Eigenschaften von Iteratoren. Damit können Sie Werte, Unterschiede, Zeiger, Referenztypen und auch die Kategorie des Iterators abrufen:

```
template<class Iter>
Iter find(Iter first, Iter last, typename std::iterator_traits<Iter>::value_type val)  {
    while (first != last) {
        if (*first == val)
            return first;
        ++first;
    }
    return last;
}
```

Die Kategorie des Iterators kann zur Spezialisierung von Algorithmen verwendet werden:

```
template<class BidirIt>
void test(BidirIt a, std::bidirectional_iterator_tag)  {
```

```
    std::cout << "Bidirectional iterator is used" << std::endl;
}

template<class ForwIt>
void test(ForwIt a, std::forward_iterator_tag)  {
    std::cout << "Forward iterator is used" << std::endl;
}

template<class Iter>
void test(Iter a)  {
    test(a, typename std::iterator_traits<Iter>::iterator_category());
}
```

Kategorien von Iteratoren sind im Wesentlichen Iteratorkonzepte, mit der Ausnahme, dass fortlaufende Iteratoren kein eigenes Tag haben, da festgestellt wurde, dass sie Code beschädigen.

Umgekehrte Iteratoren

Wenn wir durch eine Liste oder einen Vektor rückwärts iterieren möchten, können wir einen `reverse_iterator` . Ein Reverse-Iterator wird aus einem bidirektionalen oder Random-Access-Iterator erstellt, den er als Mitglied speichert, auf das über `base()` zugegriffen werden kann.

Um rückwärts zu iterieren, verwenden Sie `rbegin()` und `rend()` als Iteratoren für das Ende der Sammlung bzw. den Beginn der Sammlung.

Um zum Beispiel rückwärts zu iterieren, verwenden Sie:

```
std::vector<int> v{1, 2, 3, 4, 5};
for (std::vector<int>::reverse_iterator it = v.rbegin(); it != v.rend(); ++it)
{
    cout << *it;
} // prints 54321
```

Ein Reverse-Iterator kann über die `base()` Member-Funktion in einen Forward-Iterator konvertiert werden. Die Beziehung ist, dass der Reverse-Iterator auf ein Element hinter dem Iterator `base()` verweist:

```
std::vector<int>::reverse_iterator r = v.rbegin();
std::vector<int>::iterator i = r.base();
assert(&*r == &*(i-1)); // always true if r, (i-1) are dereferenceable
                        // and are not proxy iterators

 +---+---+---+---+---+---+---+
 |   | 1 | 2 | 3 | 4 | 5 |   |
 +---+---+---+---+---+---+---+
   ↑   ↑               ↑   ↑
   |   |               |   |
rend() |         rbegin()  end()
       |                   rbegin().base()
     begin()
     rend().base()
```

In der Visualisierung, wo Iteratoren Positionen zwischen Elementen markieren, ist die Beziehung

einfacher:

```
  +---+---+---+---+---+
  | 1 | 2 | 3 | 4 | 5 |
  +---+---+---+---+---+
  ↑                   ↑
  |                   |
  |                 end()
  |                 rbegin()
begin()             rbegin().base()
rend()
rend().base()
```

Vektor-Iterator

`begin` gibt einen `iterator` an das erste Element im Sequenzcontainer zurück.

`end` gibt einen `iterator` an das erste Element nach dem Ende zurück.

Wenn das Vektorobjekt `const` , geben sowohl `begin` als auch `end` einen `const_iterator` . Wenn Sie möchten, dass `const_iterator` zurückgegeben wird, auch wenn Ihr Vektor nicht `const` , können Sie `cbegin` und `cend` .

Beispiel:

```
#include <vector>
#include <iostream>

int main() {
    std::vector<int> v = { 1, 2, 3, 4, 5 };  //intialize vector using an initializer_list

    for (std::vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
        std::cout << *it << " ";
    }

    return 0;
}
```

Ausgabe:

```
1 2 3 4 5
```

Karten-Iterator

Ein Iterator für das erste Element im Container.

Wenn ein Kartenobjekt für const qualifiziert ist, gibt die Funktion einen `const_iterator` . Andernfalls wird ein `iterator` .

```
// Create a map and insert some values
std::map<char,int> mymap;
mymap['b'] = 100;
mymap['a'] = 200;
```

```
mymap['c'] = 300;

// Iterate over all tuples
for (std::map<char,int>::iterator it = mymap.begin(); it != mymap.end(); ++it)
    std::cout << it->first << " => " << it->second << '\n';
```

Ausgabe:

a => 200
b => 100
c => 300

Stream-Iteratoren

Stream-Iteratoren sind nützlich, wenn Sie eine Sequenz lesen oder formatierte Daten aus einem Container drucken müssen:

```
// Data stream. Any number of various whitespace characters will be OK.
std::istringstream istr("1\t 2     3 4");
std::vector<int> v;

// Constructing stream iterators and copying data from stream into vector.
std::copy(
    // Iterator which will read stream data as integers.
    std::istream_iterator<int>(istr),
    // Default constructor produces end-of-stream iterator.
    std::istream_iterator<int>(),
    std::back_inserter(v));

// Print vector contents.
std::copy(v.begin(), v.end(),
    //Will print values to standard output as integers delimeted by " -- ".
    std::ostream_iterator<int>(std::cout, " -- "));
```

Das Beispielprogramm druckt `1 -- 2 -- 3 -- 4 --` zur Standardausgabe.

Schreiben Sie Ihren eigenen Iterator mit Generatorunterstützung

Ein weit verbreitetes Muster in anderen Sprachen ist die Funktion, die einen "Strom" von Objekten erzeugt und in der Lage ist, Schleifencode zum Schleifen darüber zu verwenden.

Wir können dies in C ++ als modellieren

```
template<class T>
struct generator_iterator {
  using difference_type=std::ptrdiff_t;
  using value_type=T;
  using pointer=T*;
  using reference=T;
  using iterator_category=std::input_iterator_tag;
  std::optional<T> state;
  std::function< std::optional<T>() > operation;
  // we store the current element in "state" if we have one:
  T operator*() const {
```

```
    return *state;
  }
  // to advance, we invoke our operation.  If it returns a nullopt
  // we have reached the end:
  generator_iterator& operator++() {
    state = operation();
    return *this;
  }
  generator_iterator operator++(int) {
    auto r = *this;
    ++(*this);
    return r;
  }
  // generator iterators are only equal if they are both in the "end" state:
  friend bool operator==( generator_iterator const& lhs, generator_iterator const& rhs ) {
    if (!lhs.state && !rhs.state) return true;
    return false;
  }
  friend bool operator!=( generator_iterator const& lhs, generator_iterator const& rhs ) {
    return !(lhs==rhs);
  }
  // We implicitly construct from a std::function with the right signature:
  generator_iterator( std::function< std::optional<T>() > f ):operation(std::move(f))
  {
    if (operation)
      state = operation();
  }
  // default all special member functions:
  generator_iterator( generator_iterator && ) =default;
  generator_iterator( generator_iterator const& ) =default;
  generator_iterator& operator=( generator_iterator && ) =default;
  generator_iterator& operator=( generator_iterator const& ) =default;
  generator_iterator() =default;
};
```

Live-Beispiel .

Wir speichern das generierte Element frühzeitig, sodass wir leichter erkennen können, ob wir bereits am Ende sind.

Da die Funktion eines End-Generator-Iterators nie verwendet wird, können wir eine Reihe von Generator-Iteratoren erstellen, indem Sie die `std::function` nur einmal kopieren. Ein standardmäßig erstellter Generator-Iterator vergleicht gleich mit sich selbst und mit allen anderen End-Generator-Iteratoren.

Iteratoren online lesen: https://riptutorial.com/de/cplusplus/topic/473/iteratoren

Kapitel 64: Klassen / Strukturen

Syntax

- Variable.Mitgliedervariable = konstant;
- Variable.Mitgliedsfunktion ();
- variable_pointer-> member_var = konstant;
- variable_pointer-> member_function ();

Bemerkungen

Beachten Sie, dass der ***einzige*** Unterschied zwischen den Schlüsselwörtern `struct` und `class` besteht, dass standardmäßig die Member-Variablen, Member-Funktionen und Basisklassen einer `struct` `public` , während sie in einer `class` `private` . C ++ - Programmierer neigen dazu, sie Klasse zu nennen, wenn sie Konstruktoren und Destruktoren hat und die Möglichkeit hat, ihre eigenen Invarianten durchzusetzen. oder eine struct, wenn es sich nur um eine einfache Sammlung von Werten handelt, die C ++ - Sprache selbst unterscheidet jedoch nicht.

Examples

Klassengrundlagen

Eine *Klasse* ist ein benutzerdefinierter Typ. Eine Klasse wird mit dem Schlüsselwort `class` , `struct` oder `union` . In der umgangssprachlichen Verwendung bezieht sich der Begriff "Klasse" normalerweise nur auf nicht gewerkschaftliche Klassen.

Eine Klasse ist eine Sammlung von *Klassenmitgliedern. Dies* kann sein:

- Mitgliedsvariablen (auch "Felder" genannt),
- Elementfunktionen (auch "Methoden" genannt),
- Elementtypen oder Typedefs (zB "verschachtelte Klassen"),
- Elementvorlagen (beliebiger Art: Variable, Funktion, Klasse oder Aliasvorlage)

Die Schlüsselwörter `class` und `struct` , die als *Klassenschlüssel bezeichnet werden* , sind weitgehend austauschbar, mit der Ausnahme, dass der Standardzugriffsspezifizierer für Mitglieder und Basen für eine mit dem `class` deklarierte `class` "private" und für eine mit dem `struct` oder `union` Schlüssel deklarierte `class` "public" ist (siehe Zugriffsmodifizierer).

Die folgenden Codeausschnitte sind beispielsweise identisch:

```
struct Vector
{
    int x;
    int y;
    int z;
};
```

```
// are equivalent to
class Vector
{
public:
    int x;
    int y;
    int z;
};
```

Durch die Deklaration einer Klasse wird Ihrem Programm ein neuer Typ hinzugefügt, und es ist möglich, Objekte dieser Klasse durch zu instanziieren

```
Vector my_vector;
```

Auf Mitglieder einer Klasse wird mit Punktsyntax zugegriffen.

```
my_vector.x = 10;
my_vector.y = my_vector.x + 1; // my_vector.y = 11;
my_vector.z = my_vector.y - 4; // my:vector.z = 7;
```

Zugriffsspezifizierer

Es gibt drei Schlüsselwörter , die als **Zugriffsspezifizierer dienen** . Diese beschränken den Zugriff auf Klassenmitglieder nach dem Bezeichner, bis ein anderer Bezeichner die Zugriffsebene erneut ändert:

Stichwort	Beschreibung
`public`	Jeder hat Zugang
`protected`	Nur die Klasse selbst, abgeleitete Klassen und Freunde haben Zugriff
`private`	Nur die Klasse selbst und Freunde haben Zugriff

Wenn der Typ mit dem Schlüsselwort `class` definiert wird, ist der Standardzugriffsbezeichner `private` . Wenn der Typ mit dem Schlüsselwort `struct` definiert ist, ist der Standardzugriffsbezeichner `public` :

```
struct MyStruct { int x; };
class MyClass { int x; };

MyStruct s;
s.x = 9; // well formed, because x is public

MyClass c;
c.x = 9; // ill-formed, because x is private
```

Zugriffsspezifizierer werden meist verwendet, um den Zugriff auf interne Felder und Methoden zu beschränken und den Programmierer zu zwingen, eine bestimmte Schnittstelle zu verwenden, z. B. um die Verwendung von Getern und Setters zu erzwingen, anstatt eine Variable direkt zu

referenzieren:

```
class MyClass {

public: /* Methods: */

    int x() const noexcept { return m_x; }
    void setX(int const x) noexcept { m_x = x; }

private: /* Fields: */

    int m_x;

};
```

Die Verwendung von `protected` ist nützlich, um zuzulassen, dass bestimmte Funktionen des Typs nur für die abgeleiteten Klassen zugänglich sind. Beispielsweise kann im folgenden Code auf die Methode `calculateValue()` nur Klassen `Plus2Base` , die von der Basisklasse `Plus2Base` , beispielsweise `FortyTwo` :

```
struct Plus2Base {
    int value() noexcept { return calculateValue() + 2; }
protected: /* Methods: */
    virtual int calculateValue() noexcept = 0;
};
struct FortyTwo: Plus2Base {
protected: /* Methods: */
    int calculateValue() noexcept final override { return 40; }
};
```

Beachten Sie, dass das `friend` Schlüsselwort verwendet werden kann, um Zugriffsausnahmen zu Funktionen oder Typen für den Zugriff auf geschützte und private Mitglieder hinzuzufügen.

Die `public` , `protected` und `private` Schlüsselwörter können auch verwendet werden, um den Zugriff auf Basisklassen-Unterobjekte zu gewähren oder einzuschränken. Siehe das Vererbungsbeispiel .

Erbe

Klassen / Strukturen können Vererbungsbeziehungen haben.

Wenn eine Klasse / Struktur `B` von einer Klasse / Struktur `A` erbt, bedeutet dies, dass `B` übergeordnetes `A` . Wir sagen, dass `B` eine abgeleitete Klasse / Struktur von `A` ist und `A` die Basisklasse / Struktur ist.

```
struct A
{
public:
    int p1;
protected:
    int p2;
private:
    int p3;
};
```

```
//Make B inherit publicly (default) from A
struct B : A
{
};
```

Es gibt drei Arten der Vererbung für eine Klasse / Struktur:

- `public`
- `private`
- `protected`

Beachten Sie, dass die Standardvererbung der Standardsichtbarkeit von Mitgliedern entspricht: `public` wenn Sie das Schlüsselwort `struct` , und `private` für das Schlüsselwort `class` .

Es ist sogar möglich, eine `class` von einer `struct` ableiten zu lassen (oder umgekehrt). In diesem Fall wird die Standard - Vererbung durch das Kind gesteuert, so dass eine `struct` , die aus einer ableitet `class` auf öffentliche Vererbung ausfällt, und eine `class` , die von einem leitet `struct` standardmäßig private Vererbung hat.

`public` Erbe:

```
struct B : public A // or just `struct B : A`
{
    void foo()
    {
        p1 = 0; //well formed, p1 is public in B
        p2 = 0; //well formed, p2 is protected in B
        p3 = 0; //ill formed, p3 is private in A
    }
};

B b;
b.p1 = 1; //well formed, p1 is public
b.p2 = 1; //ill formed, p2 is protected
b.p3 = 1; //ill formed, p3 is inaccessible
```

`private` Vererbung:

```
struct B : private A
{
    void foo()
    {
        p1 = 0; //well formed, p1 is private in B
        p2 = 0; //well formed, p2 is private in B
        p3 = 0; //ill formed, p3 is private in A
    }
};

B b;
b.p1 = 1; //ill formed, p1 is private
b.p2 = 1; //ill formed, p2 is private
b.p3 = 1; //ill formed, p3 is inaccessible
```

`protected` Vererbung:

```
struct B : protected A
{
    void foo()
    {
        p1 = 0; //well formed, p1 is protected in B
        p2 = 0; //well formed, p2 is protected in B
        p3 = 0; //ill formed, p3 is private in A
    }
};

B b;
b.p1 = 1; //ill formed, p1 is protected
b.p2 = 1; //ill formed, p2 is protected
b.p3 = 1; //ill formed, p3 is inaccessible
```

Beachten Sie, dass die `protected` Vererbung zwar zulässig ist, die tatsächliche Verwendung jedoch selten ist. Ein Beispiel dafür, wie `protected` Vererbung in der Anwendung verwendet wird, ist die teilweise Spezialisierung der Basisklasse (normalerweise als "kontrollierter Polymorphismus" bezeichnet).

Wenn OOP relativ neu war, wurde häufig gesagt, dass (öffentliche) Vererbung eine "IS-A" -Beziehung modelliert. Öffentliche Vererbung ist also nur dann korrekt, wenn eine Instanz der abgeleiteten Klasse *auch eine* Instanz der Basisklasse ist.

Dies wurde später im Liskov-Substitutionsprinzip verfeinert: Öffentliche Vererbung sollte nur verwendet werden, wenn / wenn eine Instanz der abgeleiteten Klasse unter allen möglichen Umständen (und immer noch sinnvoll) für eine Instanz der Basisklasse eingesetzt werden kann.

Private Vererbung soll typischerweise eine völlig andere Beziehung verkörpern: "wird in Form von" implementiert (manchmal als "HAS-A" -Beziehung bezeichnet). Eine `Stack` Klasse kann beispielsweise privat von einer `Vector` Klasse erben. Private Vererbung hat eine größere Ähnlichkeit mit der Aggregation als mit der öffentlichen Vererbung.

Geschützte Vererbung wird fast nie verwendet, und es besteht keine generelle Übereinstimmung darüber, welche Art von Beziehung sie beinhaltet.

Virtuelle Vererbung

Bei der Vererbung können Sie das `virtual` Schlüsselwort angeben:

```
struct A{};
struct B: public virtual A{};
```

Wenn die Klasse `B` über die virtuelle Basis `A` , bedeutet dies, dass sich `A` **in der am meisten abgeleiteten Klasse** des Vererbungsbaums befindet und somit auch die am meisten abgeleitete Klasse für die Initialisierung dieser virtuellen Basis verantwortlich ist:

```
struct A
{
    int member;
    A(int param)
    {
```

```
        member = param;
    }
};

struct B: virtual A
{
    B(): A(5){}
};

struct C: B
{
    C(): /*A(88)*/ {}
};

void f()
{
    C object; //error since C is not initializing it's indirect virtual base `A`
}
```

Wenn wir `/*A(88)*/` unkommentieren, erhalten wir keine Fehlermeldung, da `C` jetzt seine indirekte virtuelle Basis `A` initialisiert.

Beachten Sie auch , dass , wenn wir variabel sind die Schaffung `object` , die meisten abgeleitete Klasse ist `C` , so `C` verantwortlich ist für die Erstellung von (Aufruf Konstruktor) `A` und somit Wert von `A::member` ist `88` , nicht `5` (wie es wäre, wenn wir waren Objekt vom Typ `B`) `B` .

Es ist nützlich, um das Diamantproblem zu lösen .

```
  A                                    A   A
 / \                                   |   |
B   C                                  B   C
 \ /                                    \ /
  D                                      D
virtual inheritance                  normal inheritance
```

`B` und `C` erben beide von `A` und `D` erbt von `B` und `C` , also **gibt es 2 Fälle von `A` in `D` !** Dies führt zu Mehrdeutigkeiten, wenn Sie auf Member von `A` bis `D` zugreifen, da der Compiler nicht wissen kann, von welcher Klasse Sie auf dieses Member zugreifen möchten (diejenige, die `B` erbt oder die von `C` geerbt wird?). .

Die virtuelle Vererbung löst dieses Problem: Da sich die virtuelle Basis nur in den meisten abgeleiteten Objekten befindet, gibt es nur eine Instanz von `A` in `D`

```
struct A
{
    void foo() {}
};

struct B : public /*virtual*/ A {};
struct C : public /*virtual*/ A {};

struct D : public B, public C
{
    void bar()
    {
```

```
        foo(); //Error, which foo? B::foo() or C::foo()? - Ambiguous
    }
};
```

Durch das Entfernen der Kommentare wird die Mehrdeutigkeit behoben.

Mehrfachvererbung

Abgesehen von der einfachen Vererbung:

```
class A {};
class B : public A {};
```

Sie können auch mehrere Vererbungen haben:

```
class A {};
class B {};
class C : public A, public B {};
```

`C` wird jetzt gleichzeitig von `A` und `B` erben.

Hinweis: Dies kann zu Mehrdeutigkeiten führen, wenn in mehreren geerbten `class` oder `struct` dieselben Namen verwendet werden. Achtung!

Mehrdeutigkeit bei der Mehrfachvererbung

Mehrfachvererbung kann in bestimmten Fällen hilfreich sein, aber manchmal kommt es zu merkwürdigen Problemen bei der Verwendung von Mehrfachvererbung.

Beispiel: Zwei Basisklassen haben Funktionen mit demselben Namen, die in der abgeleiteten Klasse nicht überschrieben werden. Wenn Sie Code schreiben, um auf diese Funktion mithilfe des Objekts der abgeleiteten Klasse zuzugreifen, zeigt der Compiler einen Fehler an, da er nicht ermitteln kann, welche Funktion aufgerufen werden soll. Hier ist ein Code für diese Art der Mehrdeutigkeit bei der Mehrfachvererbung.

```
class base1
{
  public:
     void funtion( )
     { //code for base1 function }
};
class base2
{
    void function( )
     { // code for base2 function }
};

class derived : public base1, public base2
{

};

int main()
```

```
{
    derived obj;

  // Error because compiler can't figure out which function to call
  //either function( ) of base1 or base2 .
    obj.function( )
}
```

Dieses Problem kann jedoch durch Verwendung der Bereichsauflösungsfunktion gelöst werden, um anzugeben, welche Funktion entweder base1 oder base2 klassifiziert:

```
int main()
{
    obj.base1::function( );  // Function of class base1 is called.
    obj.base2::function( );  // Function of class base2 is called.
}
```

Zugriff auf die Klassenmitglieder

Um auf Member-Variablen und Member-Funktionen eines Objekts einer Klasse zuzugreifen, muss der `.` Operator wird verwendet:

```
struct SomeStruct {
  int a;
  int b;
  void foo() {}
};

SomeStruct var;
// Accessing member variable a in var.
std::cout << var.a << std::endl;
// Assigning member variable b in var.
var.b = 1;
// Calling a member function.
var.foo();
```

Beim Zugriff auf die Member einer Klasse über einen Zeiger wird häufig der Operator `->` verwendet. Alternativ kann die Instanz dereferenziert werden und die `.` Operator verwendet, obwohl dies weniger üblich ist:

```
struct SomeStruct {
  int a;
  int b;
  void foo() {}
};

SomeStruct var;
SomeStruct *p = &var;
// Accessing member variable a in var via pointer.
std::cout << p->a << std::endl;
std::cout << (*p).a << std::endl;
// Assigning member variable b in var via pointer.
p->b = 1;
(*p).b = 1;
// Calling a member function via a pointer.
```

```
p->foo();
(*p).foo();
```

Beim Zugriff auf statische Klassenmitglieder wird der Operator `::` verwendet, jedoch auf den Namen der Klasse und nicht auf eine Instanz davon. Alternativ kann auf das statische Member von einer Instanz oder von einem Zeiger auf eine Instanz mit der Option zugegriffen werden `.` oder `->` Operator mit derselben Syntax wie der Zugriff auf nicht statische Member.

```
struct SomeStruct {
  int a;
  int b;
  void foo() {}

  static int c;
  static void bar() {}
};
int SomeStruct::c;

SomeStruct var;
SomeStruct* p = &var;
// Assigning static member variable c in struct SomeStruct.
SomeStruct::c = 5;
// Accessing static member variable c in struct SomeStruct, through var and p.
var.a = var.c;
var.b = p->c;
// Calling a static member function.
SomeStruct::bar();
var.bar();
p->bar();
```

Hintergrund

Der Operator `->` wird benötigt, weil der Member-Zugriffsoperator `.` hat Vorrang vor dem Dereferenzierungsoperator `*` .

Man würde erwarten, dass `*pa` deneference `p` (was dazu führt, dass auf das Objekt, auf das `p` zeigt, verweist) und dann auf sein Element `a` zugreift. Tatsächlich versucht es jedoch, auf das Mitglied `a` von `p` zuzugreifen und es dann zu dereferenzieren. Dh `*pa` entspricht `*(pa)` . Im obigen Beispiel würde dies aus zwei Gründen zu einem Compiler-Fehler führen: Erstens ist `p` ein Zeiger und hat kein Member `a` . Zweitens ist `a` eine ganze Zahl und kann daher nicht dereferenziert werden.

Die ungewöhnlich verwendete Lösung für dieses Problem wäre die explizite Steuerung der Priorität: `(*p).a`

Stattdessen wird fast immer der Operator `->` verwendet. Es ist eine Abkürzung, um den Zeiger zuerst dereferenzieren und dann darauf zugreifen zu können. `(*p).a` ist genau dasselbe wie `p->a` .

Der Operator `::` ist der Bereichsoperator, der auf dieselbe Weise wie beim Zugriff auf ein Member eines Namespaces verwendet wird. Dies liegt daran, dass ein statischer Klassenmitglied als im Gültigkeitsbereich dieser Klasse enthalten gilt, jedoch nicht als Mitglied von Instanzen dieser Klasse betrachtet wird. Die Verwendung von normalen `.` und `->` ist aus statischen Gründen auch

für statische Mitglieder zulässig, obwohl sie keine Instanzmitglieder sind; Dies ist für das Schreiben von generischem Code in Vorlagen von Nutzen, da der Aufrufer sich nicht darum kümmern muss, ob eine bestimmte Elementfunktion statisch oder nicht statisch ist.

Private Vererbung: Einschränkung der Basisklassenschnittstelle

Die private Vererbung ist nützlich, wenn die öffentliche Schnittstelle der Klasse eingeschränkt werden muss:

```
class A {
public:
    int move();
    int turn();
};

class B : private A {
public:
    using A::turn;
};

B b;
b.move();  // compile error
b.turn();  // OK
```

Dieser Ansatz verhindert effizient den Zugriff auf die öffentlichen A-Methoden, indem er auf den A-Zeiger oder die A-Referenz angewendet wird:

```
B b;
A& a = static_cast<A&>(b); // compile error
```

Im Falle der öffentlichen Vererbung bietet ein solches Casting Zugang zu allen A-öffentlichen Methoden, obwohl es alternative Methoden gibt, um dies in abgeleitetem B zu verhindern.

```
class B : public A {
private:
    int move();
};
```

oder privat mit:

```
class B : public A {
private:
    using A::move;
};
```

dann ist es in beiden Fällen möglich:

```
B b;
A& a = static_cast<A&>(b); // OK for public inheritance
a.move(); // OK
```

Abschlussklassen und -strukturen

C ++ 11

Das Ableiten einer Klasse kann mit dem `final` Bezeichner verboten werden. Lassen Sie uns eine letzte Klasse erklären:

```
class A final {
};
```

Jeder Versuch der Unterklasse führt zu einem Kompilierungsfehler:

```
// Compilation error: cannot derive from final class:
class B : public A {
};
```

Die letzte Klasse kann an einer beliebigen Stelle in der Klassenhierarchie erscheinen:

```
class A {
};

// OK.
class B final : public A {
};

// Compilation error: cannot derive from final class B.
class C : public B {
};
```

Freundschaft

Das `friend` Schlüsselwort wird verwendet, um anderen Klassen und Funktionen Zugriff auf private und geschützte Mitglieder der Klasse zu gewähren, auch wenn diese außerhalb des Bereichs der Klasse definiert sind.

```
class Animal{
private:
    double weight;
    double height;
public:
    friend void printWeight(Animal animal);
    friend class AnimalPrinter;
    // A common use for a friend function is to overload the operator<< for streaming.
    friend std::ostream& operator<<(std::ostream& os, Animal animal);
};

void printWeight(Animal animal)
{
    std::cout << animal.weight << "\n";
}

class AnimalPrinter
{
public:
```

```
    void print(const Animal& animal)
    {
        // Because of the `friend class AnimalPrinter;" declaration, we are
        // allowed to access private members here.
        std::cout << animal.weight << ", " << animal.height << std::endl;
    }
}

std::ostream& operator<<(std::ostream& os, Animal animal)
{
    os << "Animal height: " << animal.height << "\n";
    return os;
}

int main() {
    Animal animal = {10, 5};
    printWeight(animal);

    AnimalPrinter aPrinter;
    aPrinter.print(animal);

    std::cout << animal;
}
```

```
10
10, 5
Animal height: 5
```

Verschachtelte Klassen / Strukturen

Eine `class` oder `struct` kann auch eine andere `class` / `struct` enthalten, die als "verschachtelte Klasse" bezeichnet wird. In dieser Situation wird die enthaltende Klasse als "einschließende Klasse" bezeichnet. Die geschachtelte Klassendefinition wird als Mitglied der umgebenden Klasse betrachtet, ist jedoch ansonsten getrennt.

```
struct Outer {
    struct Inner { };
};
```

Von außerhalb der umgebenden Klasse wird auf verschachtelte Klassen mit dem Bereichsoperator zugegriffen. Innerhalb der umgebenden Klasse können jedoch geschachtelte Klassen ohne Qualifikationsmerkmale verwendet werden:

```
struct Outer {
    struct Inner { };

    Inner in;
};

// ...

Outer o;
Outer::Inner i = o.in;
```

Wie bei einer nicht verschachtelten `class` / `struct` können `struct` und statische Variablen entweder innerhalb einer verschachtelten Klasse oder im umschließenden Namespace definiert werden. Sie können jedoch nicht innerhalb der umgebenden Klasse definiert werden, da sie als andere Klasse als die verschachtelte Klasse betrachtet wird.

```
// Bad.
struct Outer {
    struct Inner {
        void do_something();
    };

    void Inner::do_something() {}
};


// Good.
struct Outer {
    struct Inner {
        void do_something();
    };

};

void Outer::Inner::do_something() {}
```

Wie bei nicht geschachtelten Klassen können verschachtelte Klassen später deklariert und definiert werden, sofern sie vor ihrer direkten Verwendung definiert werden.

```
class Outer {
    class Inner1;
    class Inner2;

    class Inner1 {};

    Inner1 in1;
    Inner2* in2p;

  public:
    Outer();
    ~Outer();
};

class Outer::Inner2 {};

Outer::Outer() : in1(Inner1()), in2p(new Inner2) {}
Outer::~Outer() {
    if (in2p) { delete in2p; }
}
```

C ++ 11

Vor C ++ 11 hatten verschachtelte Klassen nur Zugriff auf Typnamen, `static` Member und Enumeratoren der umgebenden Klasse. Alle anderen Mitglieder, die in der umgebenden Klasse definiert wurden, waren gesperrt.

C ++ 11

Ab C ++ 11 werden verschachtelte Klassen und deren Mitglieder so behandelt, als wären sie `friend` der umgebenden Klasse, und sie können gemäß den üblichen Zugriffsregeln auf alle ihre Mitglieder zugreifen. Wenn Mitglieder der geschachtelten Klasse die Fähigkeit benötigen, ein oder mehrere nicht statische Mitglieder der umgebenden Klasse auszuwerten, müssen sie einer Instanz übergeben werden:

```
class Outer {
    struct Inner {
        int get_sizeof_x() {
            return sizeof(x); // Legal (C++11): x is unevaluated, so no instance is required.
        }

        int get_x() {
            return x; // Illegal: Can't access non-static member without an instance.
        }

        int get_x(Outer& o) {
            return o.x; // Legal (C++11): As a member of Outer, Inner can access private
members.
        }
    };

    int x;
};
```

Umgekehrt wird die umgebende Klasse *nicht* als Freund der verschachtelten Klasse behandelt und kann daher nicht auf ihre privaten Mitglieder zugreifen, ohne dass ihnen explizit die Erlaubnis erteilt wurde.

```
class Outer {
    class Inner {
        // friend class Outer;

        int x;
    };

    Inner in;

  public:
    int get_x() {
        return in.x; // Error: int Outer::Inner::x is private.
        // Uncomment "friend" line above to fix.
    }
};
```

Freunde einer verschachtelten Klasse werden nicht automatisch als Freunde der umgebenden Klasse betrachtet. Wenn sie auch Freunde der umgebenden Klasse sein müssen, muss dies separat angegeben werden. Da die umschließende Klasse nicht automatisch als Freund der verschachtelten Klasse angesehen wird, werden Freunde der umschließenden Klasse auch nicht als Freunde der verschachtelten Klasse betrachtet.

```
class Outer {
    friend void barge_out(Outer& out, Inner& in);

    class Inner {
```

```
        friend void barge_in(Outer& out, Inner& in);

        int i;
    };

    int o;
};

void barge_in(Outer& out, Outer::Inner& in) {
    int i = in.i;  // Good.
    int o = out.o; // Error: int Outer::o is private.
}

void barge_out(Outer& out, Outer::Inner& in) {
    int i = in.i;  // Error: int Outer::Inner::i is private.
    int o = out.o; // Good.
}
```

Wie bei allen anderen Klassenmitgliedern können verschachtelte Klassen nur von außerhalb der Klasse benannt werden, wenn sie öffentlichen Zugriff haben. Sie können jedoch unabhängig vom Zugriffsmodifizierer auf sie zugreifen, sofern Sie sie nicht explizit benennen.

```
class Outer {
    struct Inner {
        void func() { std::cout << "I have no private taboo.\n"; }
    };

  public:
    static Inner make_Inner() { return Inner(); }
};

// ...

Outer::Inner oi; // Error: Outer::Inner is private.

auto oi = Outer::make_Inner(); // Good.
oi.func();                     // Good.
Outer::make_Inner().func();    // Good.
```

Sie können auch einen Typalias für eine verschachtelte Klasse erstellen. Wenn ein Typalias in der umgebenden Klasse enthalten ist, können der verschachtelte Typ und der Typalias unterschiedliche Zugriffsmodifizierer haben. Wenn sich der `typedef` außerhalb der umgebenden Klasse befindet, muss die geschachtelte Klasse oder ein `typedef` davon öffentlich sein.

```
class Outer {
    class Inner_ {};

  public:
    typedef Inner_ Inner;
};

typedef Outer::Inner  ImOut; // Good.
typedef Outer::Inner_ ImBad; // Error.

// ...
```

```
Outer::Inner  oi; // Good.
Outer::Inner_ oi; // Error.
ImOut         oi; // Good.
```

Wie bei anderen Klassen können auch verschachtelte Klassen von anderen Klassen abgeleitet oder abgeleitet werden.

```
struct Base {};

struct Outer {
    struct Inner : Base {};
};

struct Derived : Outer::Inner {};
```

Dies kann in Situationen hilfreich sein, in denen die umgebende Klasse von einer anderen Klasse abgeleitet wird, indem der Programmierer die geschachtelte Klasse bei Bedarf aktualisieren kann. Dies kann mit einem Typedef kombiniert werden, um einen konsistenten Namen für die verschachtelten Klassen jeder umgebenden Klasse bereitzustellen:

```
class BaseOuter {
    struct BaseInner_ {
        virtual void do_something() {}
        virtual void do_something_else();
    } b_in;

  public:
    typedef BaseInner_ Inner;

    virtual ~BaseOuter() = default;

    virtual Inner& getInner() { return b_in; }
};

void BaseOuter::BaseInner_::do_something_else() {}

// ---

class DerivedOuter : public BaseOuter {
    // Note the use of the qualified typedef; BaseOuter::BaseInner_ is private.
    struct DerivedInner_ : BaseOuter::Inner {
        void do_something() override {}
        void do_something_else() override;
    } d_in;

  public:
    typedef DerivedInner_ Inner;

    BaseOuter::Inner& getInner() override { return d_in; }
};

void DerivedOuter::DerivedInner_::do_something_else() {}

// ...

// Calls BaseOuter::BaseInner_::do_something();
```

```
BaseOuter* b = new BaseOuter;
BaseOuter::Inner& bin = b->getInner();
bin.do_something();
b->getInner().do_something();

// Calls DerivedOuter::DerivedInner_::do_something();
BaseOuter* d = new DerivedOuter;
BaseOuter::Inner& din = d->getInner();
din.do_something();
d->getInner().do_something();
```

In dem obigen Fall geben `BaseOuter` und `DerivedOuter` den `DerivedOuter Inner` als `BaseInner_` bzw. `DerivedInner_` an. Dadurch können geschachtelte Typen abgeleitet werden, ohne die Schnittstelle der umgebenden Klasse zu beschädigen, und der geschachtelte Typ kann polymorph verwendet werden.

Mitgliedstypen und Aliase

Eine `class` oder `struct` kann auch Mitgliedstyp-Aliasnamen definieren, die in der Klasse selbst enthalten sind und als Mitglieder dieser Klasse behandelt werden.

```
struct IHaveATypedef {
    typedef int MyTypedef;
};

struct IHaveATemplateTypedef {
    template<typename T>
    using MyTemplateTypedef = std::vector<T>;
};
```

Auf diese Typedefs wird wie statische Member mit dem Bereichsoperator `::` zugegriffen.

```
IHaveATypedef::MyTypedef i = 5; // i is an int.

IHaveATemplateTypedef::MyTemplateTypedef<int> v; // v is a std::vector<int>.
```

Wie bei normalen Typ-Aliasnamen darf jeder Mitgliedstyp-Alias vor jeder Definition, aber nicht nach ihrer Definition, auf jeden definierten Typ oder Alias verweisen. Ebenso kann ein Typedef außerhalb der Klassendefinition auf alle zugänglichen Typedefs innerhalb der Klassendefinition verweisen, sofern diese hinter der Klassendefinition stehen.

```
template<typename T>
struct Helper {
    T get() const { return static_cast<T>(42); }
};

struct IHaveTypedefs {
//    typedef MyTypedef NonLinearTypedef; // Error if uncommented.
    typedef int MyTypedef;
    typedef Helper<MyTypedef> MyTypedefHelper;
};

IHaveTypedefs::MyTypedef        i; // x_i is an int.
IHaveTypedefs::MyTypedefHelper hi; // x_hi is a Helper<int>.
```

```
typedef IHaveTypedefs::MyTypedef TypedefBeFree;
TypedefBeFree ii;                  // ii is an int.
```

Aliasnamen für Mitgliedstypen können mit jeder Zugriffsebene deklariert werden und berücksichtigen den entsprechenden Zugriffsmodifizierer.

```
class TypedefAccessLevels {
    typedef int PrvInt;

  protected:
    typedef int ProInt;

  public:
    typedef int PubInt;
};

TypedefAccessLevels::PrvInt prv_i; // Error: TypedefAccessLevels::PrvInt is private.
TypedefAccessLevels::ProInt pro_i; // Error: TypedefAccessLevels::ProInt is protected.
TypedefAccessLevels::PubInt pub_i; // Good.

class Derived : public TypedefAccessLevels {
    PrvInt prv_i; // Error: TypedefAccessLevels::PrvInt is private.
    ProInt pro_i; // Good.
    PubInt pub_i; // Good.
};
```

Dies kann verwendet werden, um eine Abstraktionsebene bereitzustellen, die es einem Designer der Klasse ermöglicht, seine internen Funktionen zu ändern, ohne den Code zu beschädigen, der darauf beruht.

```
class Something {
    friend class SomeComplexType;

    short s;
    // ...

  public:
    typedef SomeComplexType MyHelper;

    MyHelper get_helper() const { return MyHelper(8, s, 19.5, "shoe", false); }

    // ...
};

// ...

Something s;
Something::MyHelper hlp = s.get_helper();
```

Wenn in dieser Situation die `SomeComplexType` von `SomeComplexType` in einen anderen Typ geändert wird, müssen nur die `typedef` und die `friend` Deklaration geändert werden. Solange die `Something::MyHelper` die gleiche Funktionalität bietet, funktioniert jeder Code, der sie als `Something::MyHelper` anstatt sie nach Namen anzugeben, in der Regel ohne Änderungen. Auf diese Weise minimieren wir die Menge an Code, die geändert werden muss, wenn die zugrunde

liegende Implementierung geändert wird, sodass der Typname nur an einem Ort geändert werden muss.

Dies kann auch mit `decltype` kombiniert `decltype` , wenn Sie dies `decltype` .

```
class SomethingElse {
    AnotherComplexType<bool, int, SomeThirdClass> helper;

  public:
    typedef decltype(helper) MyHelper;

  private:
    InternalVariable<MyHelper> ivh;

    // ...

  public:
    MyHelper& get_helper() const { return helper; }

    // ...
};
```

In dieser Situation ändert das Ändern der Implementierung von `SomethingElse::helper` automatisch den Typedef für uns aufgrund von `decltype` . Dies reduziert die Anzahl der erforderlichen Änderungen, wenn der `helper` , wodurch das Risiko menschlicher Fehler minimiert wird.

Wie bei allem kann dies jedoch zu weit gehen. Wenn der Typname beispielsweise nur einmal oder zweimal intern und null Mal außerhalb verwendet wird, muss kein Alias angegeben werden. Wenn das Projekt hunderte oder tausende Male in einem Projekt verwendet wird oder einen ausreichend langen Namen hat, kann es nützlich sein, es als Typedef anzugeben, anstatt es immer in absoluten Zahlen zu verwenden. Kompatibilität und Bequemlichkeit müssen mit der Menge an unnötigem Rauschen in Einklang gebracht werden.

Dies kann auch mit Vorlagenklassen verwendet werden, um den Zugriff auf die Vorlagenparameter von außerhalb der Klasse zu ermöglichen.

```
template<typename T>
class SomeClass {
    // ...

  public:
    typedef T MyParam;
    MyParam getParam() { return static_cast<T>(42); }
};

template<typename T>
typename T::MyParam some_func(T& t) {
    return t.getParam();
}

SomeClass<int> si;
int i = some_func(si);
```

Dies wird im Allgemeinen bei Containern verwendet, die normalerweise ihren Elementtyp und

andere Hilfstypen als Aliase für Mitgliedstypen angeben. Die meisten Container in der C ++ - Standardbibliothek enthalten beispielsweise die folgenden 12 Hilfstypen sowie alle anderen speziellen Typen, die sie benötigen.

```
template<typename T>
class SomeContainer {
    // ...

  public:
    // Let's provide the same helper types as most standard containers.
    typedef T                                     value_type;
    typedef std::allocator<value_type>            allocator_type;
    typedef value_type&                           reference;
    typedef const value_type&                     const_reference;
    typedef value_type*                           pointer;
    typedef const value_type*                     const_pointer;
    typedef MyIterator<value_type>                iterator;
    typedef MyConstIterator<value_type>           const_iterator;
    typedef std::reverse_iterator<iterator>       reverse_iterator;
    typedef std::reverse_iterator<const_iterator> const_reverse_iterator;
    typedef size_t                                size_type;
    typedef ptrdiff_t                             difference_type;
};
```

Vor C ++ 11 wurde häufig auch eine Art "Template- `typedef` " angegeben, da die Funktion noch nicht verfügbar war. Diese sind mit der Einführung von Alias-Templates etwas seltener geworden, sie sind jedoch in einigen Situationen immer noch nützlich (und werden in anderen Situationen mit Alias-Templates kombiniert), was sehr nützlich sein kann, um einzelne Komponenten eines komplexen Typs, z). Sie verwenden üblicherweise den `type` für ihren Typalias.

```
template<typename T>
struct TemplateTypedef {
    typedef T type;
}

TemplateTypedef<int>::type i; // i is an int.
```

Dies wurde häufig bei Typen mit mehreren Vorlagenparametern verwendet, um einen Aliasnamen bereitzustellen, der einen oder mehrere Parameter definiert.

```
template<typename T, size_t SZ, size_t D>
class Array { /* ... */ };

template<typename T, size_t SZ>
struct OneDArray {
    typedef Array<T, SZ, 1> type;
};

template<typename T, size_t SZ>
struct TwoDArray {
    typedef Array<T, SZ, 2> type;
};

template<typename T>
struct MonoDisplayLine {
    typedef Array<T, 80, 1> type;
```

```
};

OneDArray<int, 3>::type     arr1i; // arr1i is an Array<int, 3, 1>.
TwoDArray<short, 5>::type   arr2s; // arr2s is an Array<short, 5, 2>.
MonoDisplayLine<char>::type arr3c; // arr3c is an Array<char, 80, 1>.
```

Statische Klassenmitglieder

Eine Klasse darf auch `static` Member haben, die entweder Variablen oder Funktionen sein können. Diese werden als innerhalb des Bereichs der Klasse betrachtet, aber nicht als normale Mitglieder behandelt. Sie haben eine statische Speicherdauer (sie existieren vom Beginn des Programms bis zum Ende), sind nicht an eine bestimmte Instanz der Klasse gebunden, und für die gesamte Klasse ist nur eine Kopie vorhanden.

```
class Example {
    static int num_instances;      // Static data member (static member variable).
    int i;                         // Non-static member variable.

  public:
    static std::string static_str; // Static data member (static member variable).
    static int static_func();      // Static member function.

    // Non-static member functions can modify static member variables.
    Example() { ++num_instances; }
    void set_str(const std::string& str);
};

int         Example::num_instances;
std::string Example::static_str = "Hello.";

// ...

Example one, two, three;
// Each Example has its own "i", such that:
//  (&one.i != &two.i)
//  (&one.i != &three.i)
//  (&two.i != &three.i).
// All three Examples share "num_instances", such that:
//  (&one.num_instances == &two.num_instances)
//  (&one.num_instances == &three.num_instances)
//  (&two.num_instances == &three.num_instances)
```

Statische Member-Variablen werden nicht als innerhalb der Klasse definiert, sondern nur als deklariert betrachtet und haben daher ihre Definition außerhalb der Klassendefinition. Der Programmierer darf statische Variablen in ihrer Definition initialisieren, ist aber nicht dazu verpflichtet. Bei der Definition der Elementvariablen wird das Schlüsselwort `static` weggelassen.

```
class Example {
    static int num_instances;              // Declaration.

  public:
    static std::string static_str;         // Declaration.

    // ...
};
```

```
int         Example::num_instances;         // Definition.  Zero-initialised.
std::string Example::static_str = "Hello."; // Definition.
```

Aus diesem Grund können statische Variablen (abgesehen von `void`) unvollständige Typen sein, sofern sie später als vollständiger Typ definiert werden.

```
struct ForwardDeclared;

class ExIncomplete {
    static ForwardDeclared fd;
    static ExIncomplete    i_contain_myself;
    static int             an_array[];
};

struct ForwardDeclared {};

ForwardDeclared ExIncomplete::fd;
ExIncomplete    ExIncomplete::i_contain_myself;
int             ExIncomplete::an_array[5];
```

Statische Memberfunktionen können wie bei normalen Memberfunktionen innerhalb oder außerhalb der Klassendefinition definiert werden. Wie bei statischen Elementvariablen wird das Schlüsselwort `static` bei der Definition statischer Elementfunktionen außerhalb der Klassendefinition weggelassen.

```
// For Example above, either...
class Example {
    // ...

  public:
    static int static_func() { return num_instances; }

    // ...

    void set_str(const std::string& str) { static_str = str; }
};

// Or...

class Example { /* ... */ };

int  Example::static_func() { return num_instances; }
void Example::set_str(const std::string& str) { static_str = str; }
```

Wenn eine statische Membervariable als `const` deklariert wird, aber nicht `volatile` ist und vom Integral- oder Aufzählungstyp ist, kann sie bei der Deklaration innerhalb der Klassendefinition initialisiert werden.

```
enum E { VAL = 5 };

struct ExConst {
    const static int ci = 5;             // Good.
    static const E ce = VAL;             // Good.
```

```
    const static double cd = 5;           // Error.
    static const volatile int cvi = 5;    // Error.

    const static double good_cd;
    static const volatile int good_cvi;
};

const double ExConst::good_cd = 5;        // Good.
const volatile int ExConst::good_cvi = 5; // Good.
```

C ++ 11

Ab C ++ 11 können statische Membervariablen von `LiteralType` Typen (Typen, die zur Kompilierzeit gemäß den `constexpr` Regeln erstellt werden können) auch als `constexpr` deklariert werden. Wenn ja, müssen sie innerhalb der Klassendefinition initialisiert werden.

```
struct ExConstexpr {
    constexpr static int ci = 5;                      // Good.
    static constexpr double cd = 5;                   // Good.
    constexpr static int carr[] = { 1, 1, 2 };        // Good.
    static constexpr ConstexprConstructibleClass c{}; // Good.
    constexpr static int bad_ci;                      // Error.
};

constexpr int ExConstexpr::bad_ci = 5;                // Still an error.
```

Wenn eine statische Membervariable `const` oder `constexpr` *odr verwendet wird* (informell, wenn ihre Adresse verwendet wird oder einer Referenz zugewiesen ist), muss sie außerhalb der Klassendefinition noch eine separate Definition haben. Diese Definition darf keinen Initialisierer enthalten.

```
struct ExODR {
    static const int odr_used = 5;
};

// const int ExODR::odr_used;

const int* odr_user = & ExODR::odr_used; // Error; uncomment above line to resolve.
```

Da statische Member nicht an eine bestimmte Instanz gebunden sind, kann auf sie mit dem Bereichsoperator `::` . Zugegriffen werden.

```
std::string str = Example::static_str;
```

Sie können auch so aufgerufen werden, als wären sie normale, nicht statische Member. Dies ist von historischer Bedeutung, wird jedoch weniger häufig als der Bereichsoperator verwendet, um Verwirrung darüber zu vermeiden, ob ein Member statisch oder nicht statisch ist.

```
Example ex;
std::string rts = ex.static_str;
```

Klassenmitglieder können auf statische Mitglieder zugreifen, ohne deren Gültigkeitsbereich

einzuschränken, wie bei nicht statischen Klassenmitgliedern.

```
class ExTwo {
    static int num_instances;
    int my_num;

  public:
    ExTwo() : my_num(num_instances++) {}

    static int get_total_instances() { return num_instances; }
    int get_instance_number() const { return my_num; }
};

int ExTwo::num_instances;
```

Sie können weder `mutable` noch müssen sie sein; Da sie nicht an eine bestimmte Instanz gebunden sind, hat dies keine Auswirkungen auf statische Member.

```
struct ExDontNeedMutable {
    int immuta;
    mutable int muta;

    static int i;

    ExDontNeedMutable() : immuta(-5), muta(-5) {}
};
int ExDontNeedMutable::i;

// ...

const ExDontNeedMutable dnm;
dnm.immuta = 5; // Error: Can't modify read-only object.
dnm.muta = 5;   // Good.  Mutable fields of const objects can be written.
dnm.i = 5;      // Good.  Static members can be written regardless of an instance's const-
ness.
```

Statische Member berücksichtigen Zugriffsmodifizierer ebenso wie nicht statische Member.

```
class ExAccess {
    static int prv_int;

  protected:
    static int pro_int;

  public:
    static int pub_int;
};

int ExAccess::prv_int;
int ExAccess::pro_int;
int ExAccess::pub_int;

// ...

int x1 = ExAccess::prv_int; // Error: int ExAccess::prv_int is private.
int x2 = ExAccess::pro_int; // Error: int ExAccess::pro_int is protected.
```

```
int x3 = ExAccess::pub_int; // Good.
```

Als sie auf eine bestimmte Instanz nicht gebunden sind, haben statische Elementfunktionen keinen `this` Zeiger; Aus diesem Grund können sie nicht auf nicht statische Member-Variablen zugreifen, wenn keine Instanz übergeben wird.

```
class ExInstanceRequired {
    int i;

  public:
    ExInstanceRequired() : i(0) {}

    static void bad_mutate() { ++i *= 5; }                          // Error.
    static void good_mutate(ExInstanceRequired& e) { ++e.i *= 5; } // Good.
};
```

Da `this` Zeiger nicht vorhanden sind, können ihre Adressen nicht in Zeiger-zu-Element-Funktionen gespeichert werden, sondern werden in normalen Zeiger-zu-Funktionen gespeichert.

```
struct ExPointer {
           void nsfunc() {}
    static void  sfunc() {}
};

typedef void (ExPointer::* mem_f_ptr)();
typedef void (*f_ptr)();

mem_f_ptr p_sf = &ExPointer::sfunc; // Error.
    f_ptr p_sf = &ExPointer::sfunc; // Good.
```

Da `this` Zeiger nicht vorhanden ist, können sie auch nicht `const` oder `volatile` und auch keine Ref-Qualifier haben. Sie können auch nicht virtuell sein.

```
struct ExCVQualifiersAndVirtual {
    static void   func()                {} // Good.
    static void  cfunc() const          {} // Error.
    static void  vfunc() volatile       {} // Error.
    static void cvfunc() const volatile {} // Error.
    static void  rfunc() &              {} // Error.
    static void rvfunc() &&             {} // Error.

    virtual static void vsfunc()        {} // Error.
    static virtual void svfunc()        {} // Error.
};
```

Da sie nicht an eine bestimmte Instanz gebunden sind, werden statische Elementvariablen effektiv als spezielle globale Variablen behandelt. Sie werden beim Start des Programms erstellt und beim Beenden gelöscht, unabhängig davon, ob tatsächlich Instanzen der Klasse vorhanden sind. Von jeder statischen Membervariablen ist nur eine einzige Kopie vorhanden (es sei denn, die Variable ist `thread_local` deklariert (C ++ 11 oder höher). In diesem Fall gibt es eine Kopie pro Thread).

Statische Member-Variablen haben dieselbe Verknüpfung wie die Klasse, unabhängig davon, ob

die Klasse über eine externe oder interne Verknüpfung verfügt. Lokale Klassen und unbenannte Klassen dürfen keine statischen Mitglieder haben.

Nicht statische Memberfunktionen

Eine Klasse kann über nicht statische Memberfunktionen verfügen , die einzelne Instanzen der Klasse bearbeiten.

```
class CL {
  public:
    void member_function() {}
};
```

Diese Funktionen werden wie folgt für eine Instanz der Klasse aufgerufen:

```
CL instance;
instance.member_function();
```

Sie können entweder innerhalb oder außerhalb der Klassendefinition definiert werden. Wenn sie außerhalb definiert sind, werden sie als im Klassenbereich angegeben angegeben.

```
struct ST {
    void  defined_inside() {}
    void defined_outside();
};
void ST::defined_outside() {}
```

Sie können für den Lebenslauf qualifiziert und / oder qualifiziert sein und beeinflussen, wie sie die angeforderte Instanz sehen. Die Funktion sieht die Instanz mit den angegebenen cv-qualifier (s), sofern vorhanden. Welche Version aufgerufen wird, hängt von den CV-Qualifiers der Instanz ab. Wenn es keine Version mit denselben CV-Qualifikationsmerkmalen wie die Instanz gibt, wird, falls verfügbar, eine Version mit höherem CV-Status aufgerufen.

```
struct CVQualifiers {
    void func()                   {} // 1: Instance is non-cv-qualified.
    void func() const             {} // 2: Instance is const.

    void cv_only() const volatile {}
};

CVQualifiers       non_cv_instance;
const CVQualifiers      c_instance;

non_cv_instance.func(); // Calls #1.
c_instance.func();      // Calls #2.

non_cv_instance.cv_only(); // Calls const volatile version.
c_instance.cv_only();      // Calls const volatile version.
```

C ++ 11

Member-Funktionsreferenzqualifizierer geben an, ob die Funktion für rvalue-Instanzen aufgerufen

werden soll oder nicht, und verwenden dieselbe Syntax wie Funktions-CV-Qualifikationsmerkmale.

```
struct RefQualifiers {
    void func() &  {} // 1: Called on normal instances.
    void func() && {} // 2: Called on rvalue (temporary) instances.
};

RefQualifiers rf;
rf.func();              // Calls #1.
RefQualifiers{}.func(); // Calls #2.
```

CV-Qualifier und Ref-Qualifier können bei Bedarf auch kombiniert werden.

```
struct BothCVAndRef {
    void func() const& {} // Called on normal instances.  Sees instance as const.
    void func() &&     {} // Called on temporary instances.
};
```

Sie können auch virtuell sein ; Dies ist grundlegend für den Polymorphismus und ermöglicht einer untergeordneten Klasse (n), dieselbe Schnittstelle wie die übergeordnete Klasse bereitzustellen, während sie ihre eigene Funktionalität bereitstellt.

```
struct Base {
    virtual void func() {}
};
struct Derived {
    virtual void func() {}
};

Base* bp = new Base;
Base* dp = new Derived;
bp.func(); // Calls Base::func().
dp.func(); // Calls Derived::func().
```

Weitere Informationen finden Sie hier .

Unbenannte Struktur / Klasse

Unbenannte `struct` ist erlaubt (Typ hat keinen Namen)

```
void foo()
{
    struct /* No name */ {
        float x;
        float y;
    } point;

    point.x = 42;
}
```

oder

```
struct Circle
```

```
{
    struct /* No name */ {
        float x;
        float y;
    } center; // but a member name
    float radius;
};
```

und später

```
Circle circle;
circle.center.x = 42.f;
```

aber NICHT *anonyme* `struct` (unbenannter Typ und unbenanntes Objekt)

```
struct InvalidCircle
{
    struct /* No name */ {
        float centerX;
        float centerY;
    }; // No member either.
    float radius;
};
```

Hinweis: Einige Compiler lassen *anonyme* `struct` als *Erweiterung zu* .

C ++ 11

- *lamdba* kann als eine spezielle *unbenannte* `struct` .
- `decltype` erlaubt den Typ der *unbenannten* `struct` abzurufen:

  ```
  decltype(circle.point) otherPoint;
  ```

- *unbenannte* `struct` kann Parameter der Vorlagenmethode sein:

  ```
  void print_square_coordinates()
  {
      const struct {float x; float y;} points[] = {
          {-1, -1}, {-1, 1}, {1, -1}, {1, 1}
      };

      // for range relies on `template <class T, std::size_t N> std::begin(T (&)[N])`
      for (const auto& point : points) {
          std::cout << "{" << point.x << ", " << point.y << "}\n";
      }

      decltype(points[0]) topRightCorner{1, 1};
      auto it = std::find(points, points + 4, topRightCorner);
      std::cout << "top right corner is the "
                << 1 + std::distance(points, it) << "th\n";
  }
  ```

Klassen / Strukturen online lesen: https://riptutorial.com/de/cplusplus/topic/508/klassen---

strukturen

Kapitel 65: Kompilieren und Bauen

Einführung

In C ++ geschriebene Programme müssen kompiliert werden, bevor sie ausgeführt werden können. Abhängig von Ihrem Betriebssystem steht eine Vielzahl von Compilern zur Verfügung.

Bemerkungen

Die meisten Betriebssysteme werden ohne Compiler ausgeliefert und müssen später installiert werden. Einige gängige Compiler-Optionen sind:

- GCC, die GNU Compiler Collection g ++
- clang: ein C-Sprachfamilien-Frontend für LLVM clang ++
- MSVC, Microsoft Visual C ++ (in Visual Studio enthalten) Visual-C ++
- C ++ Builder, Embarcadero C ++ Builder (in RAD Studio enthalten) C ++ Builder

Bitte konsultieren Sie das entsprechende Compiler-Handbuch, um ein C ++ - Programm zu erstellen.

Eine andere Möglichkeit, einen bestimmten Compiler mit einem eigenen spezifischen Build-System zu verwenden, ist es möglich, generische Build-Systeme das Projekt für einen bestimmten oder für den standardmäßig installierten Compiler konfigurieren zu lassen.

Examples

Kompilieren mit GCC

Unter der Annahme einer einzigen Quelldatei namens `main.cpp` den Befehl, zu kompilieren und verknüpfen eine nicht optimierte ausführbare Datei ist wie folgt (ohne Optimierung Kompilieren ist nützlich für die anfängliche Entwicklung und Debugging, obwohl `-Og` offiziell für neuere GCC - Versionen wird empfohlen).

```
g++ -o app -Wall main.cpp -O0
```

Um eine optimierte ausführbare Datei für die Verwendung in der Produktion zu erstellen, verwenden Sie eine der Optionen `-O` (siehe: `-O1` , `-O2` , `-O3` , `-Os` , `-Ofast`):

```
g++ -o app -Wall -O2 main.cpp
```

Wenn die Option -O weggelassen wird, wird standardmäßig -O0 (dh keine Optimierungen) verwendet (Angabe von -O ohne Zahl wird in -O1 aufgelöst).

Alternativ können Sie Optimierungsflags aus den `O` Gruppen (oder mehr experimentelle Optimierungen) direkt verwenden. Das folgende Beispiel erstellt mit `-O2` Optimierung plus ein Flag

aus der `-O3` Optimierungsebene:

```
g++ -o app -Wall -O2 -ftree-partial-pre main.cpp
```

Um eine plattformspezifische optimierte ausführbare Datei (zur Verwendung in der Produktion auf der Maschine mit der gleichen Architektur) zu erstellen, verwenden Sie Folgendes:

```
g++ -o app -Wall -O2 -march=native main.cpp
```

In `.\app.exe` `./app` wird eine Binärdatei erstellt, die mit `.\app.exe` unter Windows und `./app` unter Linux, Mac OS usw. ausgeführt werden kann.

Das Flag `-o` kann auch übersprungen werden. In diesem Fall erstellt GCC die Standard-Ausgabedatei `a.exe` unter Windows und `a.out` auf Unix-ähnlichen Systemen. Um eine Datei ohne Verknüpfung zu kompilieren, verwenden Sie die Option `-c` :

```
g++ -o file.o -Wall -c file.cpp
```

Dies erzeugt eine Objektdatei namens `file.o` die später mit anderen Dateien verknüpft werden kann, um eine Binärdatei zu erzeugen:

```
g++ -o app file.o otherfile.o
```

Weitere Informationen zu Optimierungsoptionen finden Sie unter gcc.gnu.org . Besonders hervorzuheben sind `-Og` (Optimierung mit Schwerpunkt Debugging-Erfahrung - empfohlen für den standardmäßigen Edit-Compile-Debug-Zyklus) und `-Ofast` (alle Optimierungen, auch wenn die strikte Einhaltung der Standards nicht beachtet wird).

Das `-Wall` Flag `-Wall` Warnungen für viele häufige Fehler und sollte immer verwendet werden. Um die Codequalität zu verbessern, wird häufig auch `-Wextra` , `-Wextra` und andere Warnflags zu verwenden, die nicht automatisch von `-Wall` und `-Wextra` .

Wenn der Code einen bestimmten C ++ - Standard erwartet, geben Sie an, welcher Standard verwendet werden soll, indem Sie das `-std=` angeben. Unterstützte Werte entsprechen dem Jahr der Fertigstellung für jede Version des ISO C ++ - Standards. Gültige Werte für das Flag `std=` sind seit GCC 6.1.0 `c++98` / `c++03` , `c++11` , `c++14` und `c++17` / `c++1z` . Werte, die durch einen Schrägstrich getrennt sind, sind gleichwertig.

```
g++ -std=c++11 <file>
```

GCC enthält einige compilerspezifische Erweiterungen, die deaktiviert werden, wenn sie mit einem durch das `-std=` angegebenen Standard in Konflikt stehen. Zum Kompilieren mit allen aktivierten Erweiterungen kann der Wert `gnu++XX` verwendet werden, wobei `XX` eines der Jahre ist, die von den oben aufgeführten `c++` Werten verwendet werden.

Der Standardstandard wird verwendet, wenn keiner angegeben wird. Für GCC-Versionen vor 6.1.0 lautet der Standardwert `-std=gnu++03` ; In GCC 6.1.0 und höher lautet der Standardwert `-std=gnu++14`

.

Beachten Sie, dass aufgrund von Fehlern in GCC das Flag `-pthread` beim Kompilieren und Verknüpfen vorhanden sein muss, damit GCC die mit C ++ 11 eingeführte C ++ - Standard-Threading-Funktionalität unterstützt, z. B. `std::thread` und `std::wait_for` . Das Auslassen bei Verwendung von Threading-Funktionen kann auf einigen Plattformen zu keinen Warnungen, aber zu ungültigen Ergebnissen führen .

Verlinkung mit Bibliotheken:

Verwenden Sie die Option `-l` , um den Bibliotheksnamen zu übergeben:

```
g++ main.cpp -lpcre2-8
#pcre2-8 is the PCRE2 library for 8bit code units (UTF-8)
```

Wenn sich die Bibliothek nicht im Standardbibliothekpfad befindet, fügen Sie den Pfad mit der Option `-L` :

```
g++ main.cpp -L/my/custom/path/ -lmylib
```

Mehrere Bibliotheken können miteinander verbunden werden:

```
g++ main.cpp -lmylib1 -lmylib2 -lmylib3
```

Wenn eine Bibliothek von einer anderen abhängig ist, platzieren Sie die abhängige Bibliothek **vor** der unabhängigen Bibliothek:

```
g++ main.cpp -lchild-lib -lbase-lib
```

Oder lassen Sie den Linker die Bestellung selbst über `--start-group` und `--end-group` (Hinweis: Dies hat erhebliche Leistungskosten zur Folge):

```
g++ main.cpp -Wl,--start-group -lbase-lib -lchild-lib -Wl,--end-group
```

Kompilieren mit Visual C ++ (Befehlszeile)

Für Programmierer, die von GCC oder Clang zu Visual Studio kommen, oder Programmierer, die im Allgemeinen mit der Befehlszeile vertraut sind, können Sie den Visual C ++ - Compiler über die Befehlszeile sowie die IDE verwenden.

Wenn Sie Ihren Code über die Befehlszeile in Visual Studio kompilieren möchten, müssen Sie zunächst die Befehlszeilenumgebung einrichten. Dies kann entweder durch Öffnen der `Visual Studio Command Prompt` von `Visual Studio Command Prompt` / `Developer Command Prompt` / `x86 Native Tools Command Prompt` / `x64 Native Tools Command Prompt` oder ähnliches (wie in Ihrer Version von Visual Studio bereitgestellt) oder über die Eingabeaufforderung durch Navigieren zu erfolgen das `VC` Unterverzeichnis des Installationsverzeichnisses des Compilers (normalerweise `\Program Files`

`(x86)\Microsoft Visual Studio x\VC` , wobei `x` die Versionsnummer ist (z. B. `10.0` für 2010 oder `14.0` für 2015) und Ausführen der `VCVARSALL` Batchdatei mit einem Befehlszeilenparameter, der hier angegeben wird.

Beachten Sie, dass Visual Studio im Gegensatz zu GCC kein Front-End für den Linker (`link.exe`) über den Compiler (`cl.exe`) `cl.exe` , sondern den Linker als separates Programm bereitstellt, das der Compiler beim `cl.exe` aufruft. `cl.exe` und `link.exe` können separat mit verschiedenen Dateien und Optionen verwendet werden, oder `cl` kann angewiesen werden, Dateien und Optionen zum `link` wenn beide Aufgaben zusammen ausgeführt werden. Alle für `cl` angegebenen Verknüpfungsoptionen werden in `link` , und Dateien, die nicht von `cl` verarbeitet werden, werden direkt an den `link` . Da dies hauptsächlich eine einfache Anleitung zum Kompilieren mit der Visual Studio-Befehlszeile ist, werden die Argumente für die `link` derzeit nicht beschrieben. Wenn Sie eine Liste benötigen, sehen Sie hier .

Beachten Sie, dass Argumente für `cl` die Groß- und Kleinschreibung berücksichtigen, Argumente für die `link` nicht.

[Bitte beachten Sie, dass einige der folgenden Beispiele die Windows-Shell-Variable "Aktuelles Verzeichnis" `%cd%` , wenn Sie absolute Pfadnamen angeben. Für alle, die mit dieser Variablen nicht vertraut sind, wird sie zum aktuellen Arbeitsverzeichnis erweitert. In der Befehlszeile ist dies das Verzeichnis, in dem Sie sich zum Zeitpunkt der Ausführung von " `cl` befanden, und das standardmäßig in der Eingabeaufforderung angegeben wird (wenn Ihre Eingabeaufforderung beispielsweise `C:\src>` lautet, ist `%cd% C:\src\`).]

Angenommen, eine einzige Quelldatei mit dem Namen " `main.cpp` im aktuellen Ordner lautet der Befehl zum Kompilieren und Verknüpfen einer nicht optimierten ausführbaren Datei (nützlich für die anfängliche Entwicklung und das Debuggen) (verwenden Sie eine der folgenden Möglichkeiten):

```
cl main.cpp
// Generates object file "main.obj".
// Performs linking with "main.obj".
// Generates executable "main.exe".

cl /Od main.cpp
// Same as above.
// "/Od" is the "Optimisation: disabled" option, and is the default when no /O is specified.
```

Wenn Sie eine zusätzliche Quelldatei "niam.cpp" in demselben Verzeichnis annehmen, verwenden Sie Folgendes:

```
cl main.cpp niam.cpp
// Generates object files "main.obj" and "niam.obj".
// Performs linking with "main.obj" and "niam.obj".
// Generates executable "main.exe".
```

Sie können auch Platzhalterzeichen verwenden, wie zu erwarten wäre:

```
cl main.cpp src\*.cpp
```

```
// Generates object file "main.obj", plus one object file for each ".cpp" file in folder
//  "%cd%\src".
// Performs linking with "main.obj", and every additional object file generated.
// All object files will be in the current folder.
// Generates executable "main.exe".
```

Um die ausführbare Datei umzubenennen oder zu verschieben, verwenden Sie eine der folgenden Möglichkeiten:

```
cl /o name main.cpp
// Generates executable named "name.exe".

cl /o folder\ main.cpp
// Generates executable named "main.exe", in folder "%cd%\folder".

cl /o folder\name main.cpp
// Generates executable named "name.exe", in folder "%cd%\folder".

cl /Fename main.cpp
// Same as "/o name".

cl /Fefolder\ main.cpp
// Same as "/o folder\".

cl /Fefolder\name main.cpp
// Same as "/o folder\name".
```

Beide `/o` und `/Fe` übergeben ihre Parameter (nennen wir sie `o-param`), `link` als `/OUT:o-param` zu `link` und `/OUT:o-param` die entsprechende Erweiterung (im Allgemeinen `.exe` oder `.dll`) an " `o-param` wenn dies erforderlich ist. Während `/o` und `/Fe` meines Wissens in der Funktionalität identisch sind, wird letzteres für Visual Studio bevorzugt. `/o` ist als veraltet markiert und scheint hauptsächlich für Programmierer vorgesehen zu sein, die mit GCC oder Clang vertraut sind.

Beachten Sie, dass das Leerzeichen zwischen `/o` und dem angegebenen Ordner und / oder Namen optional ist, es *darf jedoch kein* Leerzeichen zwischen `/Fe` und dem angegebenen Ordner und / oder Namen geben.

Um eine optimierte ausführbare Datei (für die Verwendung in der Produktion) zu erstellen, verwenden Sie Folgendes:

```
cl /O1 main.cpp
// Optimise for executable size.  Produces small programs, at the possible expense of slower
//  execution.

cl /O2 main.cpp
// Optimise for execution speed.  Produces fast programs, at the possible expense of larger
//  file size.

cl /GL main.cpp other.cpp
// Generates special object files used for whole-program optimisation, which allows CL to
//  take every module (translation unit) into consideration during optimisation.
// Passes the option "/LTCG" (Link-Time Code Generation) to LINK, telling it to call CL during
//  the linking phase to perform additional optimisations.  If linking is not performed at
this
```

```
//  time, the generated object files should be linked with "/LTCG".
// Can be used with other CL optimisation options.
```

Um schließlich eine plattformspezifische, optimierte ausführbare Datei zu erstellen (zur Verwendung auf der Maschine mit der angegebenen Architektur in der Produktion), wählen Sie die entsprechende Eingabeaufforderung oder den `VCVARSALL` Parameter für die Zielplattform aus. `link` sollte die gewünschte Plattform aus den Objektdateien erkennen; Wenn nicht, verwenden Sie die Option `/MACHINE` , um die Zielplattform explizit anzugeben.

```
// If compiling for x64, and LINK doesn't automatically detect target platform:
cl main.cpp /link /machine:X64
```

Eine der oben genannten Dateien erzeugt eine ausführbare Datei mit dem durch `/o` oder `/Fe` angegebenen Namen oder, falls keine angegeben ist, mit einem Namen, der mit der ersten Quell- oder Objektdatei identisch ist, die im Compiler angegeben wurde.

```
cl a.cpp b.cpp c.cpp
// Generates "a.exe".

cl d.obj a.cpp q.cpp
// Generates "d.exe".

cl y.lib n.cpp o.obj
// Generates "n.exe".

cl /o yo zp.obj pz.cpp
// Generates "yo.exe".
```

Um eine Datei (en) ohne Verknüpfung zu kompilieren, verwenden Sie:

```
cl /c main.cpp
// Generates object file "main.obj".
```

Dies weist `cl` an, das Programm ohne Aufruf einer `link` zu beenden, und erzeugt eine Objektdatei, die später mit anderen Dateien verknüpft werden kann, um eine Binärdatei zu erstellen.

```
cl main.obj niam.cpp
// Generates object file "niam.obj".
// Performs linking with "main.obj" and "niam.obj".
// Generates executable "main.exe".

link main.obj niam.obj
// Performs linking with "main.obj" and "niam.obj".
// Generates executable "main.exe".
```

Es gibt auch andere wertvolle Befehlszeilenparameter, die für Benutzer sehr nützlich sein könnten:

```
cl /EHsc main.cpp
```

```
// "/EHsc" specifies that only standard C++ ("synchronous") exceptions will be caught,
//  and `extern "C"` functions will not throw exceptions.
// This is recommended when writing portable, platform-independent code.

cl /clr main.cpp
// "/clr" specifies that the code should be compiled to use the common language runtime,
//  the .NET Framework's virtual machine.
// Enables the use of Microsoft's C++/CLI language in addition to standard ("native") C++,
//  and creates an executable that requires .NET to run.

cl /Za main.cpp
// "/Za" specifies that Microsoft extensions should be disabled, and code should be
//  compiled strictly according to ISO C++ specifications.
// This is recommended for guaranteeing portability.

cl /Zi main.cpp
// "/Zi" generates a program database (PDB) file for use when debugging a program, without
//  affecting optimisation specifications, and passes the option "/DEBUG" to LINK.

cl /LD dll.cpp
// "/LD" tells CL to configure LINK to generate a DLL instead of an executable.
// LINK will output a DLL, in addition to an LIB and EXP file for use when linking.
// To use the DLL in other programs, pass its associated LIB to CL or LINK when compiling
those
//  programs.

cl main.cpp /link /LINKER_OPTION
// "/link" passes everything following it directly to LINK, without parsing it in any way.
// Replace "/LINKER_OPTION" with any desired LINK option(s).
```

Für alle, die mit * nix-Systemen und / oder GCC / Clang-Systemen vertraut sind, können `cl` , `link` und andere Visual Studio-Befehlszeilen-Tools anstelle eines Schrägstrichs (wie `/c`) mit einem Bindestrich (wie `-c`) angegebene Parameter akzeptieren. Darüber hinaus erkennt Windows entweder einen Schrägstrich oder einen umgekehrten Schrägstrich als gültiges Pfadtrennzeichen. Daher können auch * nix-Pfade verwendet werden. Dies macht es einfach, einfache Compiler-Befehlszeilen von `g++` oder `clang++` in `cl` oder umgekehrt mit minimalen Änderungen zu konvertieren.

```
g++ -o app src/main.cpp
cl  -o app src/main.cpp
```

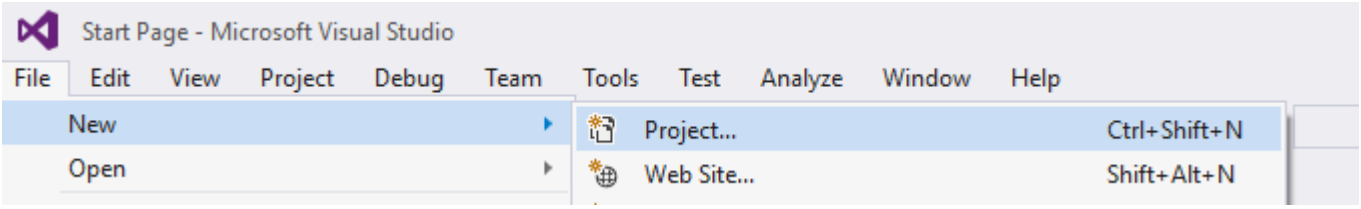
Wenn Sie Befehlszeilen portieren, die komplexere `g++` oder `clang++` Optionen verwenden, müssen Sie natürlich entsprechende Befehle in den entsprechenden Compilerdokumentationen und / oder auf Ressourcensites nachschlagen. Dies erleichtert jedoch den Einstieg mit minimalem Zeitaufwand neue Compiler.

Falls Sie für Ihren Code bestimmte Sprachfunktionen benötigen, war eine spezielle Version von MSVC erforderlich. Ab Visual C ++ 2015 Update 3 ist es möglich, die Version des Standards, mit der kompiliert werden soll, über das Flag `/std` auszuwählen. Mögliche Werte sind `/std:c++14` und `/std:c++latest` (`/std:c++17` wird in Kürze folgen).

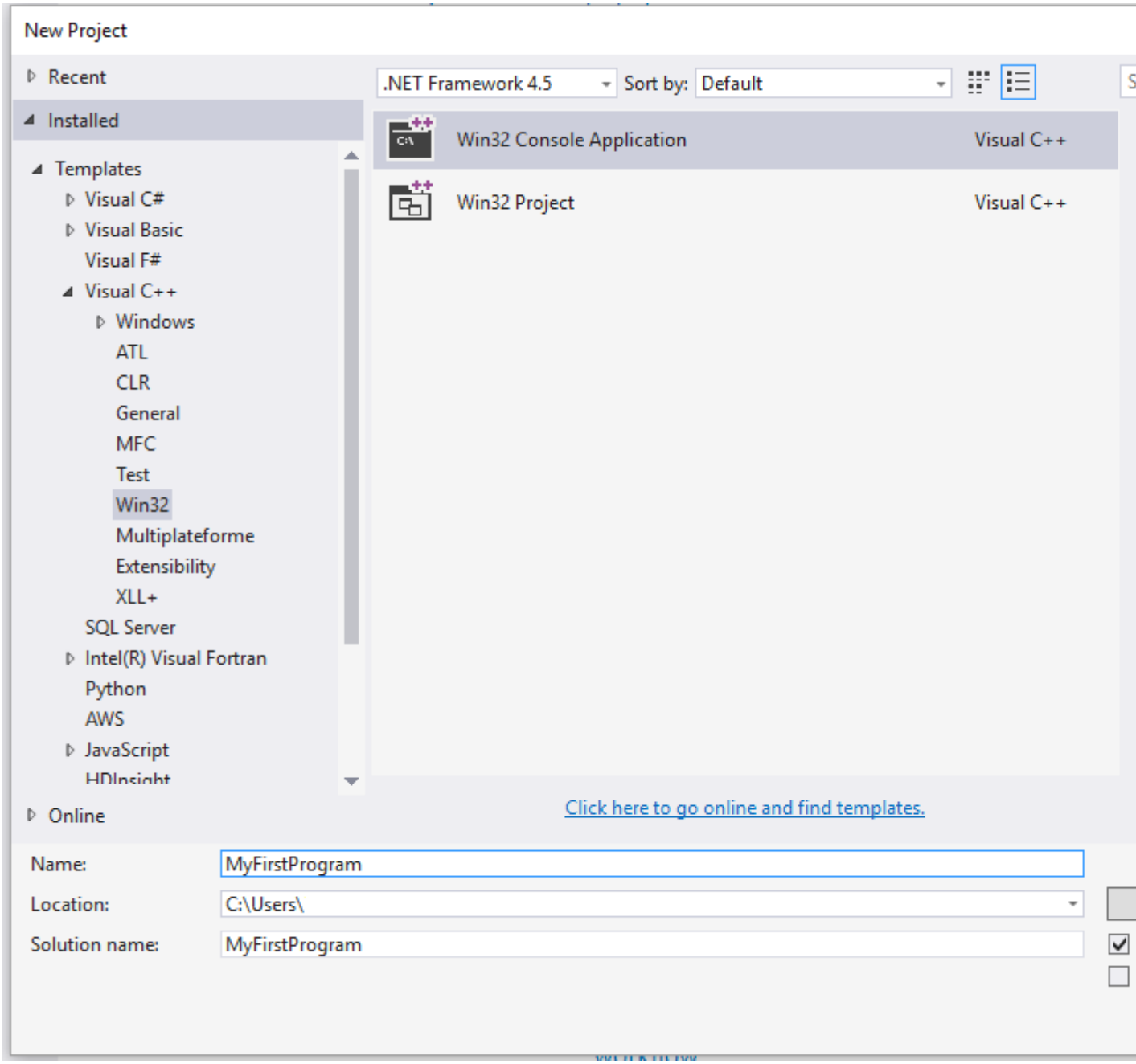
Hinweis: In älteren Versionen dieses Compilers waren bestimmte Feature-Flags verfügbar. Diese wurden jedoch hauptsächlich für die Vorschau neuer Features verwendet.

Kompilieren mit Visual Studio (grafische Benutzeroberfläche) - Hello World

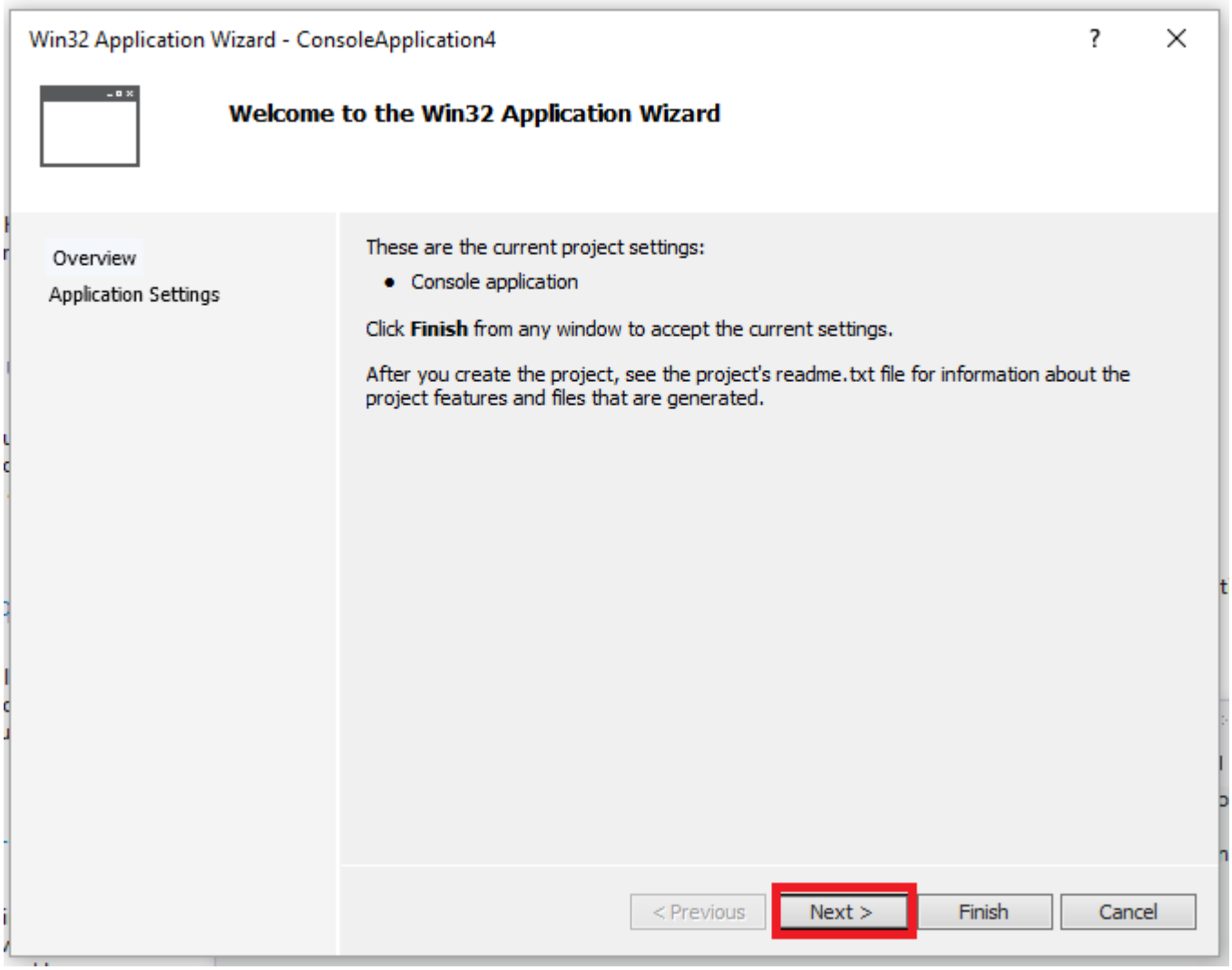
1. Laden Sie Visual Studio Community 2015 herunter und installieren Sie es
2. Öffnen Sie die Visual Studio Community
3. Klicken Sie auf Datei -> Neu -> Projekt



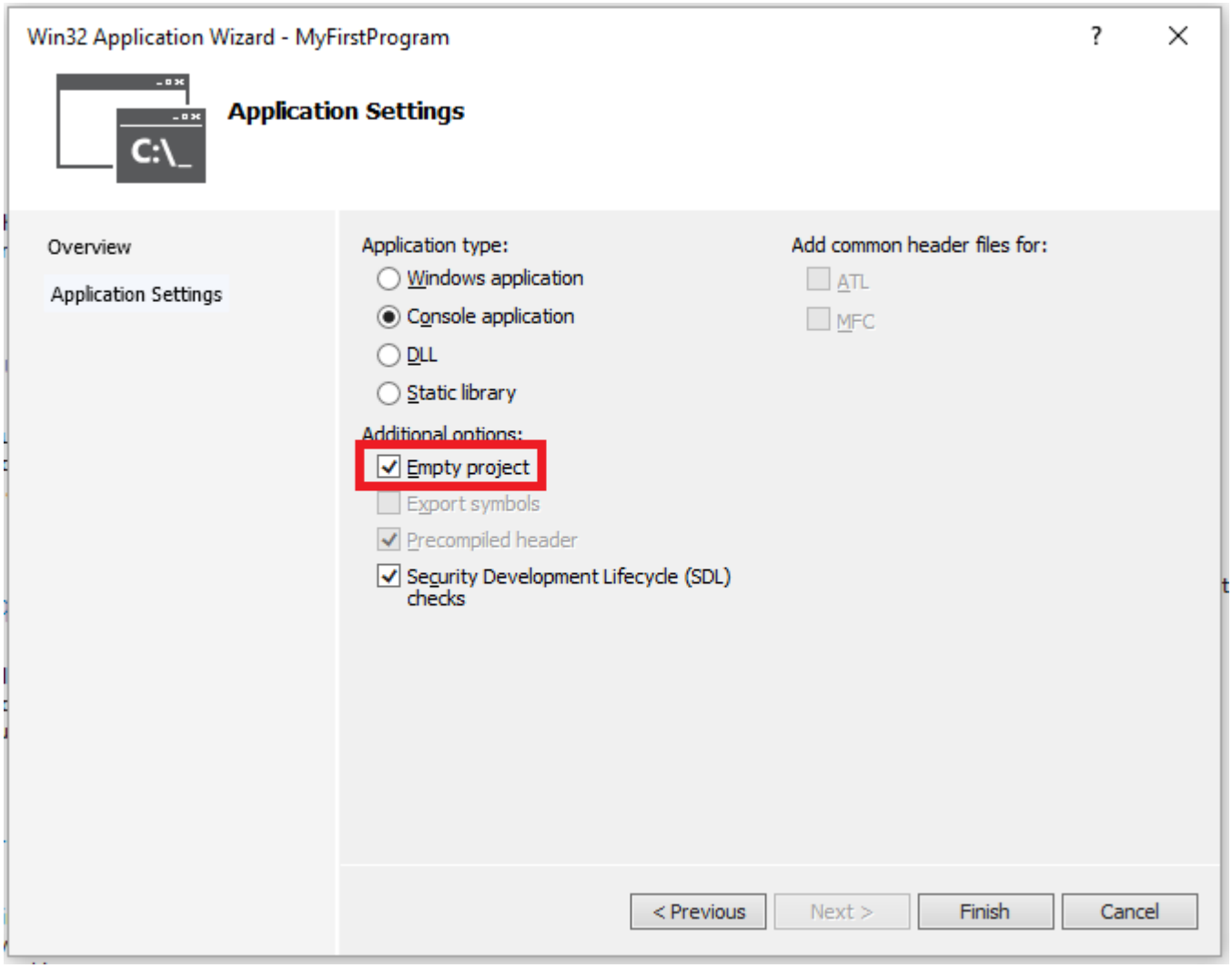
4. Klicken Sie auf Vorlagen -> Visual C ++ -> Win32-Konsolenanwendung, und nennen Sie das Projekt **MyFirstProgram** .



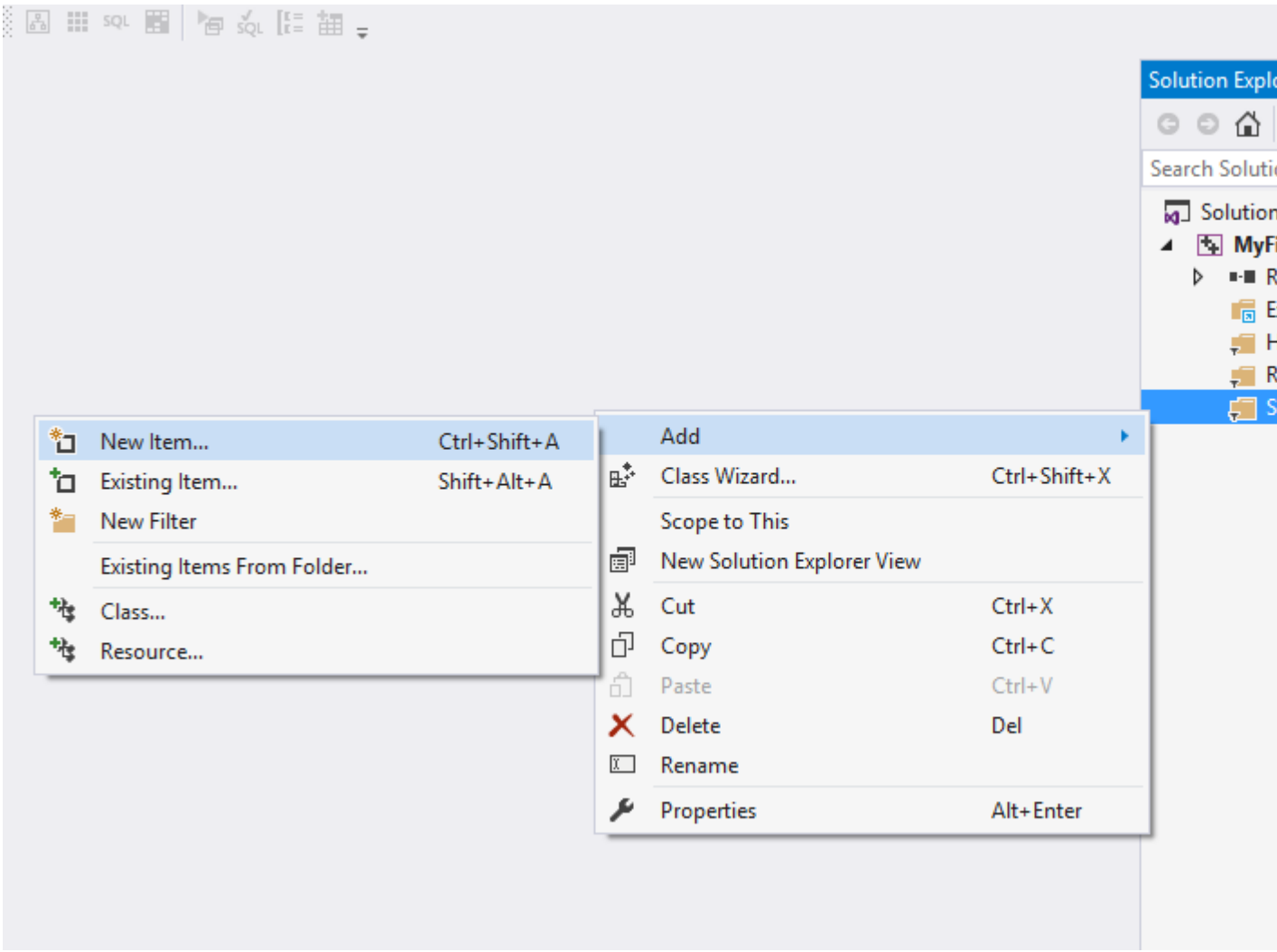
5. OK klicken
6. Klicken Sie im folgenden Fenster auf Weiter.



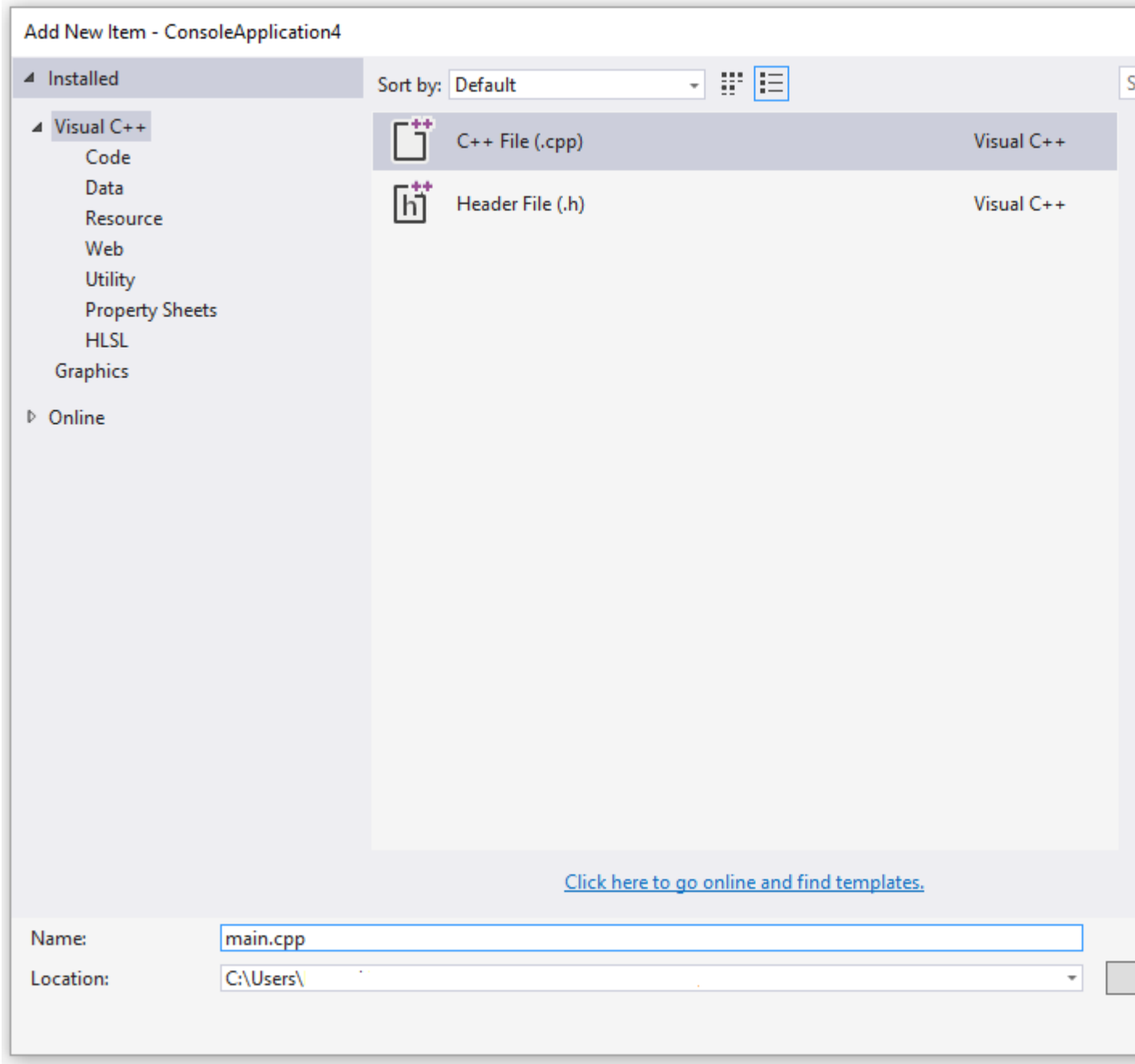
7. Aktivieren Sie das Kontrollkästchen " `Empty project` " und klicken Sie auf "Fertig stellen".



8. Klicken Sie mit der rechten Maustaste auf den Ordner Quelldatei und dann auf -> Hinzufügen -> Neues Element:



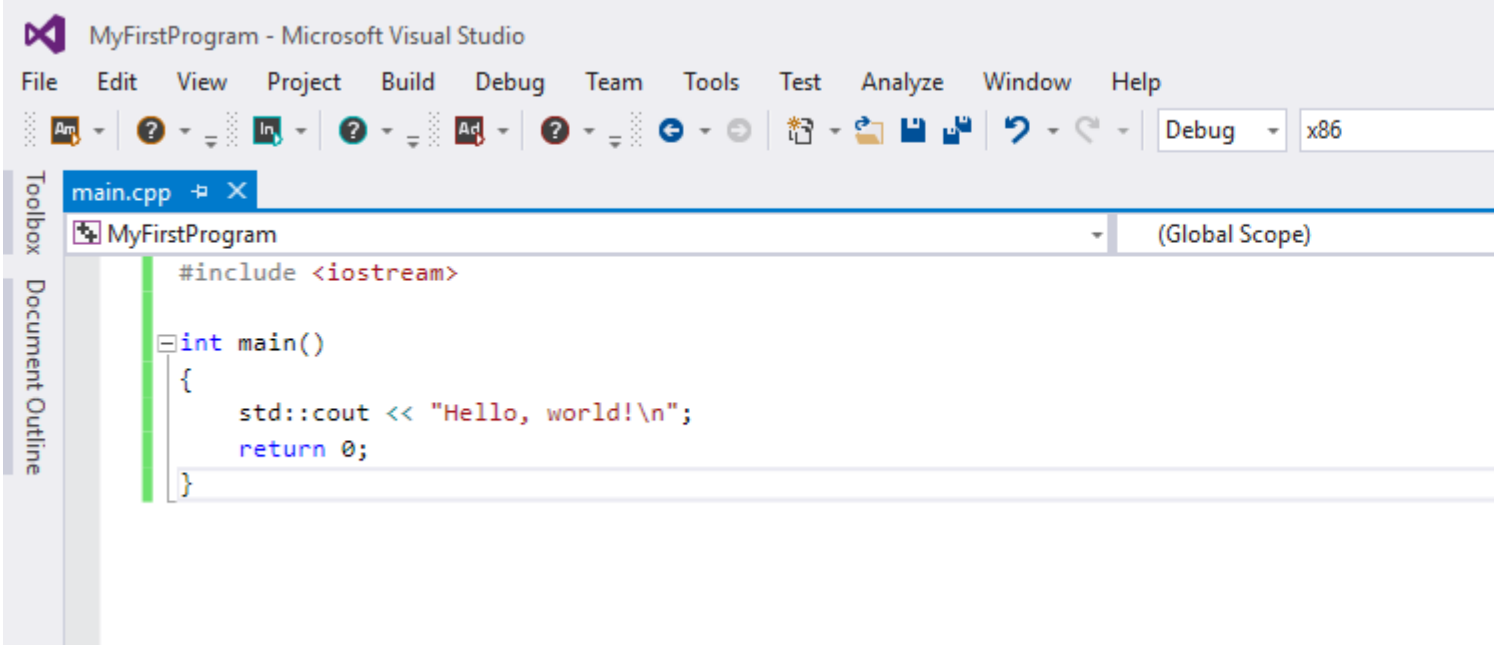
9. Wählen Sie die C ++ - Datei aus und benennen Sie die Datei main.cpp. Klicken Sie dann auf Hinzufügen:



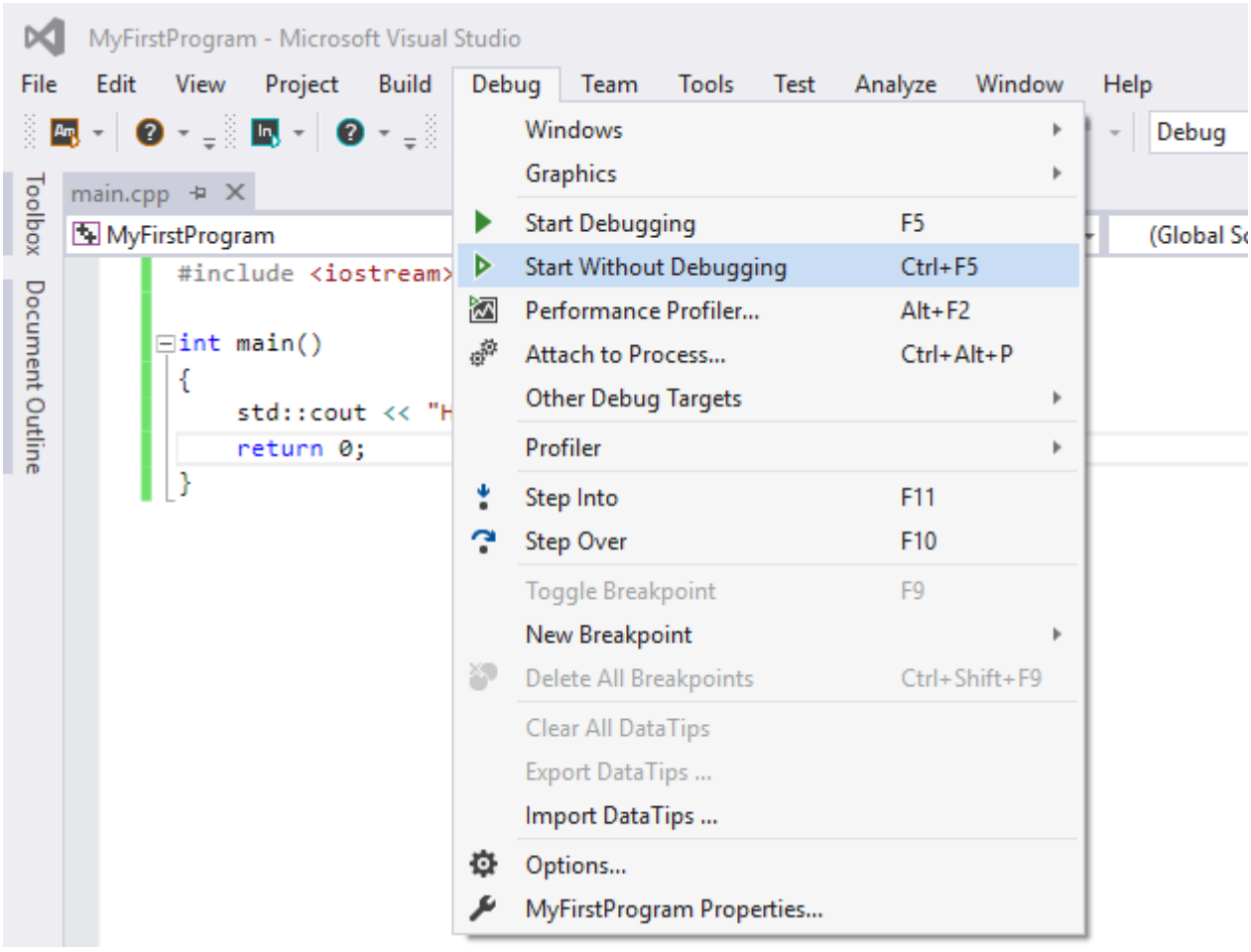
10: Kopieren Sie den folgenden Code und fügen Sie ihn in die neue Datei main.cpp ein:

```
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
    return 0;
}
```

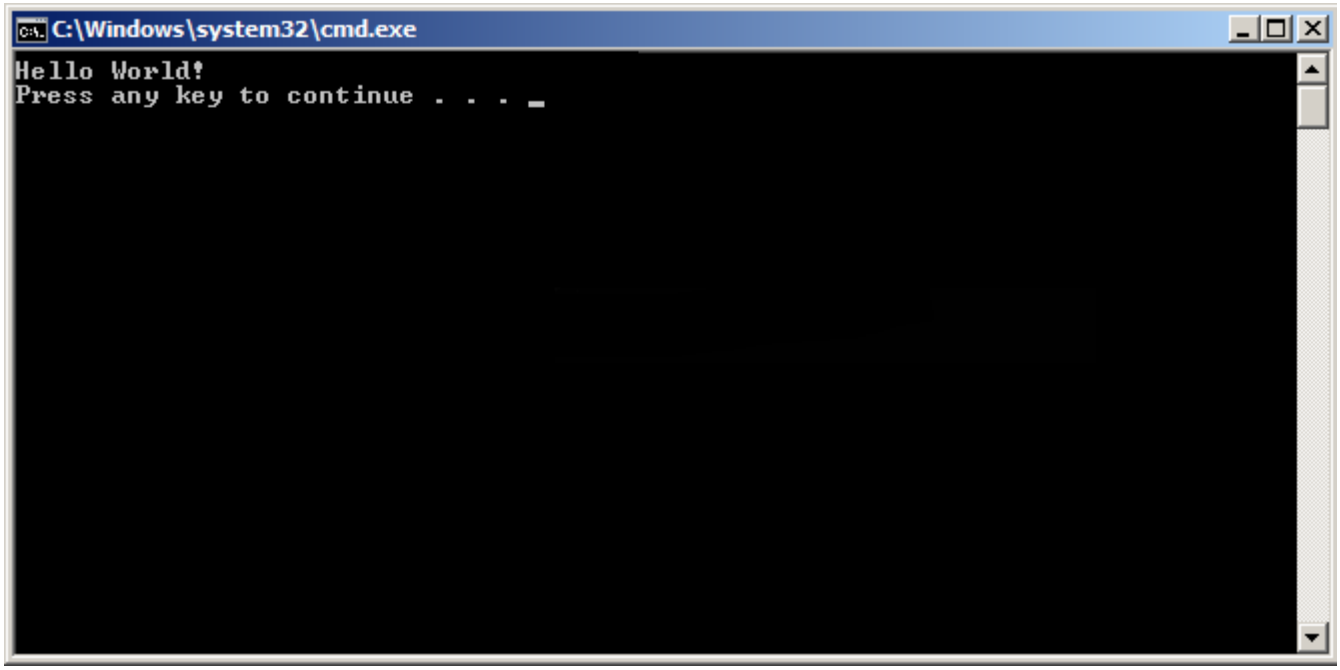
Ihre Umgebung sollte folgendermaßen aussehen:



11. Klicken Sie auf Debuggen -> Starten **ohne** Debuggen (oder drücken Sie Strg + F5):



12. Erledigt. Sie sollten die folgende Konsolenausgabe erhalten:



Kompilieren mit Clang

Da das Clang- Frontend für die Kompatibilität mit GCC ausgelegt ist, werden die meisten Programme, die über GCC kompiliert werden können, kompiliert, wenn Sie `g++` durch `clang++` in den Build-Skripts austauschen. Wenn keine `-std=version` angegeben ist, wird gnu11 verwendet.

Windows-Benutzer, die an MSVC `cl.exe` sind, können `cl.exe` mit `clang-cl.exe` . Standardmäßig versucht clang, mit der höchsten installierten Version von MSVC kompatibel zu sein.

Beim Kompilieren mit Visual Studio kann clang-cl verwendet werden, indem das `Platform toolset` in den Projekteigenschaften geändert wird.

In beiden Fällen ist clang nur über sein Frontend kompatibel, versucht jedoch auch, binärkompatible Objektdateien zu generieren. Benutzer von clang-cl sollten beachten, dass die Kompatibilität mit MSVC noch nicht vollständig ist .

Um clang oder clang-cl zu verwenden, könnte man die Standardinstallation auf bestimmten Linux-Distributionen oder mit IDEs gebündelten verwenden (wie XCode auf Mac). Für andere Versionen dieses Compilers oder auf Plattformen, auf denen dies nicht installiert ist, können Sie diese von der offiziellen Download-Seite herunterladen .

Wenn Sie CMake verwenden, um Ihren Code zu erstellen, können Sie normalerweise den Compiler wechseln, indem Sie die `CC` und `CXX` Umgebungsvariablen wie `CXX` :

```
mkdir build
cd build
CC=clang CXX=clang++ cmake ..
cmake --build .
```

Siehe auch Einführung in Cmake .

Online-Compiler

Verschiedene Websites bieten Online-Zugriff auf C ++ - Compiler. Der Funktionsumfang des Online-Compilers variiert erheblich von Website zu Site. In der Regel können Sie jedoch Folgendes tun:

- Fügen Sie Ihren Code in ein Webformular im Browser ein.
- Wählen Sie einige Compileroptionen aus und kompilieren Sie den Code.
- Sammeln Sie Compiler- und / oder Programmausgaben.

Das Verhalten der Website von Online-Compilern ist in der Regel recht restriktiv, da jeder Compiler ausführen und beliebigen Code auf der Serverseite ausführen kann, während die Ausführung von beliebigem Remote-Code normalerweise als Schwachstelle angesehen wird.

Online-Compiler können für folgende Zwecke nützlich sein:

- Führen Sie ein kleines Code-Snippet von einem Computer aus, auf dem der C ++ - Compiler fehlt (Smartphones, Tablets usw.).
- Stellen Sie sicher, dass Code erfolgreich mit verschiedenen Compilern kompiliert wird und auf dieselbe Weise ausgeführt wird, unabhängig davon, mit welchem Compiler er kompiliert wurde.
- Lernen oder lehren Sie Grundlagen von C ++.
- Lernen Sie moderne C ++ - Funktionen (in naher Zukunft C ++ 14 und C ++ 17) kennen, wenn auf dem lokalen Computer kein aktueller C ++ - Compiler verfügbar ist.
- Finden Sie einen Fehler in Ihrem Compiler im Vergleich zu einer großen Anzahl anderer Compiler. Prüfen Sie, ob in zukünftigen Versionen ein Compiler-Fehler behoben wurde, der auf Ihrem Computer nicht verfügbar ist.
- Lösen Sie Online-Richterprobleme.

Wofür Online-Compiler **nicht** verwendet werden sollten:

- Entwickeln Sie voll funktionsfähige (auch kleine) Anwendungen mit C ++. Normalerweise erlauben Online-Compiler keine Verknüpfung mit Bibliotheken von Drittanbietern oder das Herunterladen von Build-Artefakten.
- Führen Sie intensive Berechnungen durch. Die Rechenressourcen für andere Rechner sind begrenzt, so dass jedes vom Benutzer bereitgestellte Programm nach einigen Sekunden der Ausführung abgebrochen wird. Die erlaubte Ausführungszeit reicht normalerweise zum Testen und Lernen aus.
- Angriff des Compilerservers selbst oder anderer Hosts im Internet.

Beispiele:

> Haftungsausschluss: Dokumentationsautor (en) sind nicht mit den unten aufgeführten Ressourcen verbunden. Websites sind alphabetisch aufgelistet.

- http://codepad.org/ Online-Compiler mit Code-Sharing. Das Bearbeiten von Code nach dem Kompilieren mit einer Warnung oder einem Fehler im Quellcode funktioniert nicht so gut.
- http://coliru.stacked-crooked.com/ Online-Compiler, für den Sie die Befehlszeile angeben. Stellt sowohl GCC- als auch Clang-Compiler zur Verfügung.
- http://cpp.sh/ - Online-Compiler mit C ++ 14-Unterstützung. Sie können die Compiler-

Befehlszeile nicht bearbeiten. Einige Optionen sind jedoch über GUI-Steuerelemente verfügbar.

- https://gcc.godbolt.org/ - Stellt eine umfangreiche Liste von Compilerversionen, Architekturen und Disassembly-Ausgaben bereit. Sehr nützlich, wenn Sie prüfen müssen, in was Ihr Code von verschiedenen Compilern übersetzt wird. GCC, Clang, MSVC (`CL`), Intel-Compiler (`icc`), ELLCC und Zapcc sind vorhanden, wobei einer oder mehrere dieser Compiler für ARM, ARMv8 (als ARM64), Atmel AVR, MIPS, MIPS64, MSP430 und PowerPC verfügbar sind , x86- und x64-Architekturen. Compiler-Befehlszeilenargumente können bearbeitet werden.
- https://ideone.com/ - Wird im Internet häufig verwendet, um das Verhalten von Codeausschnitten darzustellen. Bietet sowohl GCC als auch Clang zur Verwendung, erlaubt jedoch nicht die Bearbeitung der Compiler-Befehlszeile.
- http://melpon.org/wandbox - Unterstützt zahlreiche Clang- und GNU / GCC-Compilerversionen.
- http://onlinegdb.com/ - Eine extrem minimalistische IDE mit einem Editor, einem Compiler (gcc) und einem Debugger (gdb).
- http://rextester.com/ - Stellt Clang-, GCC- und Visual Studio-Compiler für C und C ++ (zusammen mit Compilern für andere Sprachen) bereit, wobei die Boost-Bibliothek zur Verfügung steht.
- http://tutorialspoint.com/compile_cpp11_online.php - Eine voll funktionsfähige UNIX-Shell mit GCC und ein benutzerfreundlicher Projektexplorer.
- http://webcompiler.cloudapp.net/ - Online Visual Studio 2015-Compiler, bereitgestellt von Microsoft als Teil von RiSE4fun.

Der C ++ - Kompilierungsprozess

Wenn Sie ein C ++ - Programm entwickeln, ist der nächste Schritt das Programm zu kompilieren, bevor Sie es ausführen. Die Kompilierung ist der Prozess, der das Programm in einer von Menschen lesbaren Sprache wie C, C ++ usw. in einen Maschinencode umwandelt, der direkt von der Central Processing Unit verstanden wird. Wenn Sie beispielsweise eine C ++ - Quellcodedatei mit dem Namen prog.cpp haben und den Kompilierungsbefehl ausführen,

```
g++ -Wall -ansi -o prog prog.cpp
```

Es gibt vier Hauptschritte, um eine ausführbare Datei aus der Quelldatei zu erstellen.

1. Der C ++ - Präprozessor nimmt eine C ++ - Quellcodedatei und behandelt die Header (#include), Makros (#define) und andere Präprozessoranweisungen.

2. Die erweiterte C ++ - Quellcodedatei, die vom C ++ - Präprozessor erstellt wurde, wird in die Assembler-Sprache für die Plattform kompiliert.

3. Der vom Compiler generierte Assembler-Code wird im Objektcode für die Plattform zusammengefasst.

4. Die vom Assembler erzeugte Objektcodedatei ist miteinander verknüpft mit den Objektcodedateien für alle Bibliotheksfunktionen, die zum Erstellen einer Bibliothek oder einer ausführbaren Datei verwendet werden.

Vorverarbeitung

Der Präprozessor behandelt die Präprozessoranweisungen wie #include und #define. Es ist eine Agnostik der Syntax von C ++, weshalb es mit Vorsicht zu verwenden ist.

Es funktioniert für jeweils eine C ++ - Quelldatei, indem #include-Direktiven durch den Inhalt der jeweiligen Dateien ersetzt werden (was normalerweise nur Deklarationen ist), Makros ersetzt (#define) und verschiedene Textabschnitte in Abhängigkeit von #if ausgewählt werden. Anweisungen #ifdef und #ifndef.

Der Präprozessor arbeitet mit einem Strom von Vorverarbeitungstoken. Makrosubstitution ist definiert als Ersetzung von Token durch andere Token (der Operator ## ermöglicht das Zusammenführen von zwei Token, wenn dies sinnvoll ist).

Nach all dem erzeugt der Präprozessor eine einzige Ausgabe, die ein Tokenstrom ist, der aus den oben beschriebenen Transformationen resultiert. Außerdem werden einige spezielle Marker hinzugefügt, die dem Compiler mitteilen, woher die einzelnen Zeilen stammen, sodass er diese verwenden kann, um sinnvolle Fehlermeldungen zu erzeugen.

Einige Fehler können zu diesem Zeitpunkt unter Verwendung der Anweisungen #if und #error erzeugt werden.

Indem Sie das Compiler-Flag verwenden, können Sie den Prozess in der Vorverarbeitungsphase stoppen.

```
g++ -E prog.cpp
```

Zusammenstellung

Der Übersetzungsschritt wird an jedem Ausgang des Präprozessors durchgeführt. Der Compiler analysiert den reinen C ++ - Quellcode (jetzt ohne Präprozessoranweisungen) und konvertiert ihn in Assemblycode. Ruft dann das zugrunde liegende Back-End (Assembler in Toolchain) auf, der diesen Code zu Maschinencode zusammensetzt, um eine tatsächliche Binärdatei in einem bestimmten Format (ELF, COFF, a.out, ...) zu erstellen. Diese Objektdatei enthält den kompilierten Code (in binärer Form) der in der Eingabe definierten Symbole. Symbole in Objektdateien werden mit Namen bezeichnet.

Objektdateien können sich auf Symbole beziehen, die nicht definiert sind. Dies ist der Fall, wenn Sie eine Deklaration verwenden und keine Definition dafür angeben. Der Compiler hat nichts dagegen und wird die Objektdatei glücklich erzeugen, solange der Quellcode wohlgeformt ist.

Compiler lassen Sie normalerweise an diesem Punkt die Kompilierung abbrechen. Dies ist sehr nützlich, da Sie damit jede Quellcodedatei separat kompilieren können. Der Vorteil ist, dass Sie nicht alles neu kompilieren müssen, wenn Sie nur eine einzelne Datei ändern.

Die erzeugten Objektdateien können in speziellen Archiven, statischen Bibliotheken, abgelegt werden, um sie später wieder verwenden zu können.

In diesem Stadium werden "reguläre" Compiler-Fehler wie Syntaxfehler oder Fehler bei der

Auflösung der Überladung gemeldet.

Um den Prozess nach dem Kompilierungsschritt zu stoppen, können Sie die Option -S verwenden:

```
g++ -Wall -ansi -S prog.cpp
```

Zusammenbau

Der Assembler erstellt Objektcode. Auf einem UNIX-System werden möglicherweise Dateien mit dem Suffix .o (.OBJ auf MSDOS) angezeigt, um Objektcodedateien anzuzeigen. In dieser Phase konvertiert der Assembler diese Objektdateien vom Assemblycode in Anweisungen auf Maschinenebene. Die erstellte Datei ist ein verschiebbarer Objektcode. In der Kompilierungsphase wird daher das verschiebbare Objektprogramm generiert, und dieses Programm kann an verschiedenen Stellen verwendet werden, ohne dass es erneut kompiliert werden muss.

Um den Prozess nach dem Montageschritt zu stoppen, können Sie die Option -c verwenden:

```
g++ -Wall -ansi -c prog.cpp
```

Verlinkung

Der Linker erzeugt die endgültige Kompilierungsausgabe aus den vom Assembler erzeugten Objektdateien. Diese Ausgabe kann entweder eine gemeinsam genutzte (oder dynamische) Bibliothek sein (und obwohl der Name ähnlich ist, haben sie mit den zuvor erwähnten statischen Bibliotheken nicht viel gemeinsam) oder einer ausführbaren Datei.

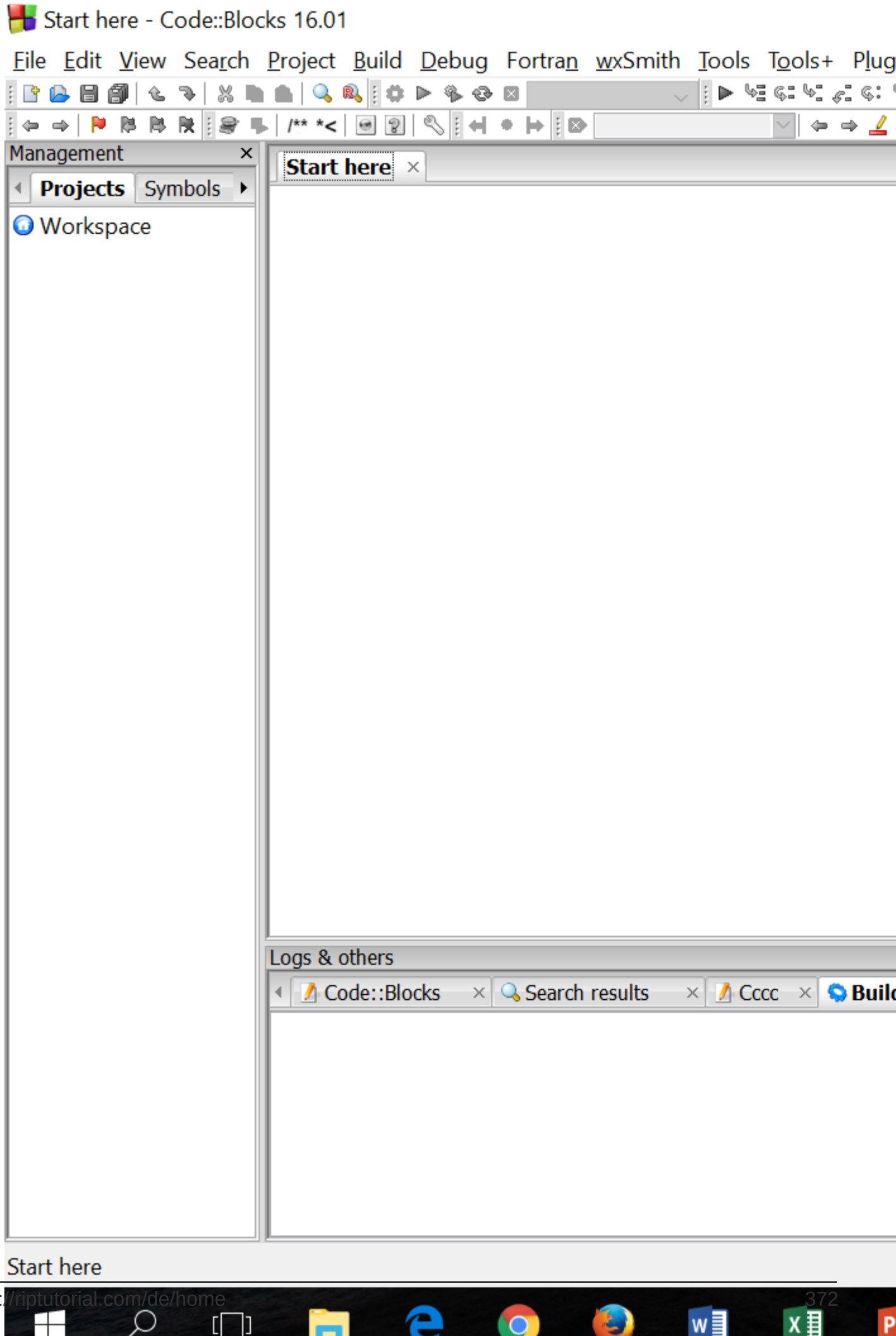
Es verknüpft alle Objektdateien, indem die Verweise auf undefinierte Symbole durch die richtigen Adressen ersetzt werden. Jedes dieser Symbole kann in anderen Objektdateien oder in Bibliotheken definiert werden. Wenn sie in anderen Bibliotheken als der Standardbibliothek definiert sind, müssen Sie den Linker darüber informieren.

Zu diesem Zeitpunkt sind die häufigsten Fehler fehlende Definitionen oder doppelte Definitionen. Ersteres bedeutet, dass entweder die Definitionen nicht existieren (dh sie werden nicht geschrieben) oder dass die Objektdateien oder Bibliotheken, in denen sie sich befinden, nicht an den Linker übergeben wurden. Letzteres ist offensichtlich: Das gleiche Symbol wurde in zwei verschiedenen Objektdateien oder Bibliotheken definiert.

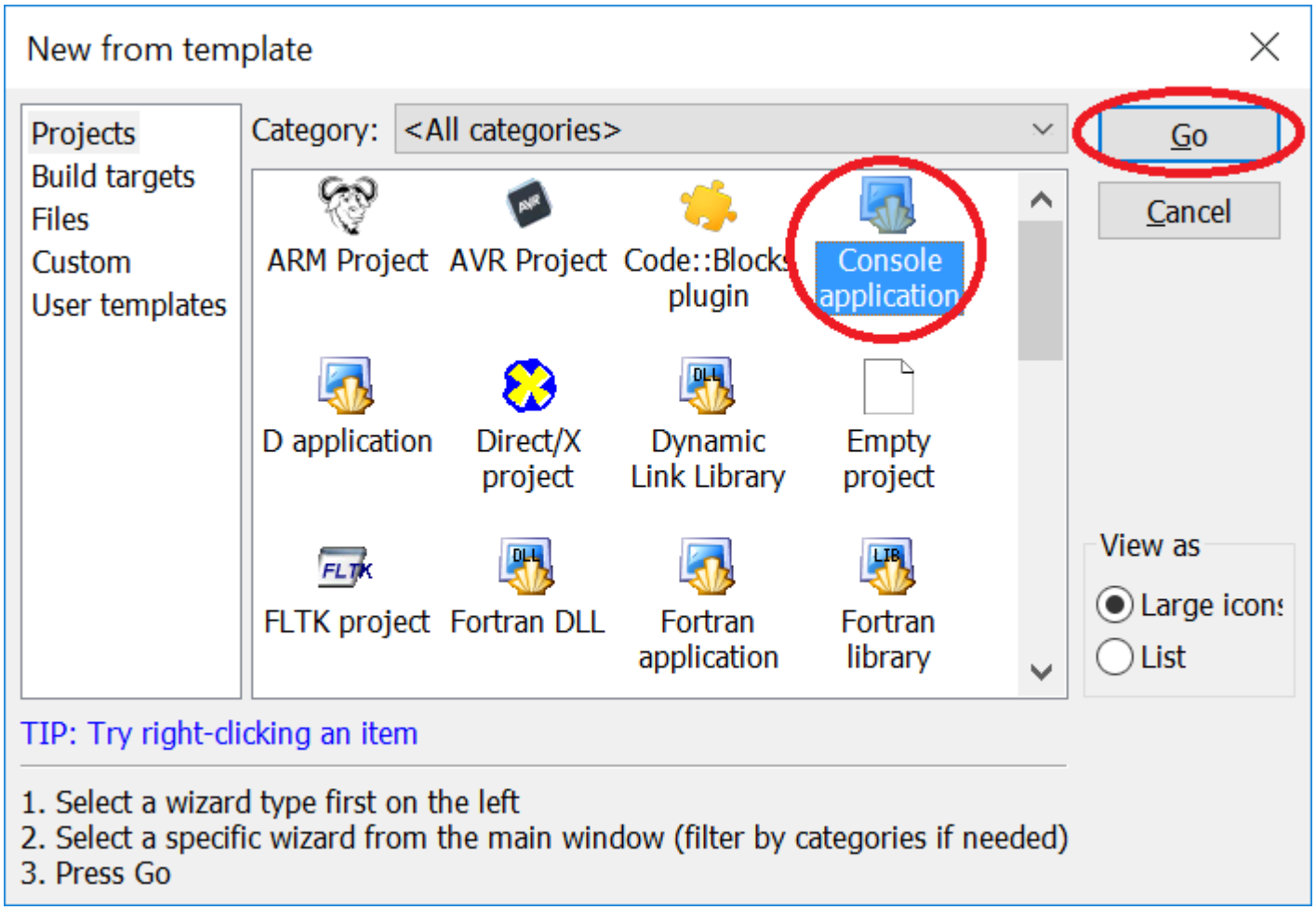
Kompilieren mit Code :: Blocks (grafische Oberfläche)

1. Laden Sie Code :: Blocks hier herunter und installieren Sie sie. Wenn Sie unter Windows sind, wählen Sie eine Datei aus, für die der Name `mingw` enthält. Die anderen Dateien installieren keinen Compiler.

2. Öffnen Sie Code :: Blocks und klicken Sie auf "Neues Projekt erstellen":

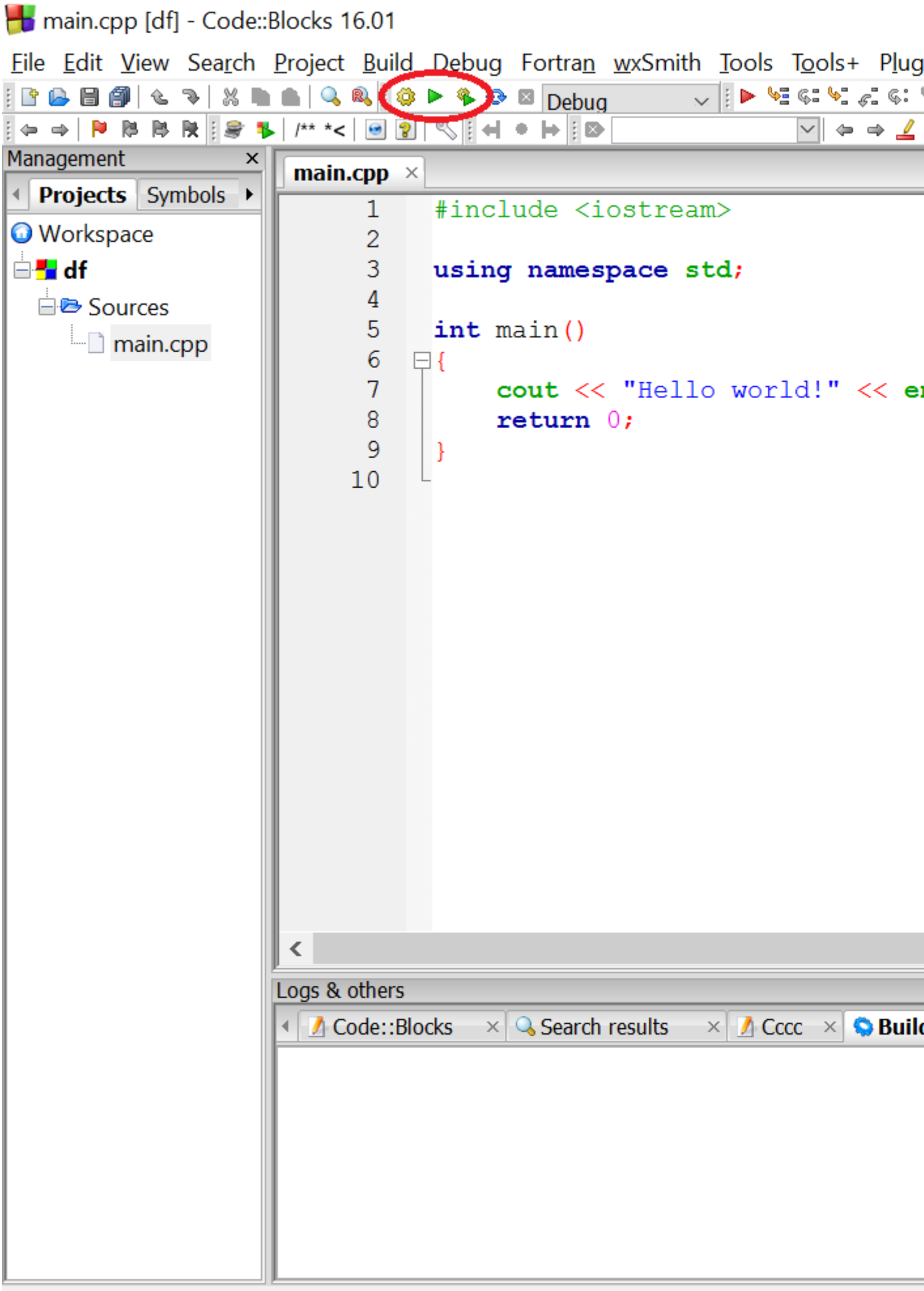

Start here - Code::Blocks 16.01
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plug
Management
Projects
Symbols
Workspace
Start here
Logs & others
Code::Blocks
Search results
Cccc
Start here

3. Wählen Sie "Konsolenanwendung" und klicken Sie auf "Los":



4. Klicken Sie auf "Weiter", wählen Sie "C ++" aus, klicken Sie auf "Weiter", wählen Sie einen Namen für Ihr Projekt und wählen Sie einen Ordner zum Speichern aus, klicken Sie auf "Weiter" und klicken Sie dann auf "Fertig stellen".

5. Jetzt können Sie Ihren Code bearbeiten und kompilieren. Ein Standardcode, der "Hallo Welt!" in der Konsole ist schon da. Um Ihr Programm zu kompilieren und / oder auszuführen, drücken Sie eine der drei Schaltflächen zum Kompilieren / Ausführen in der Symbolleiste:

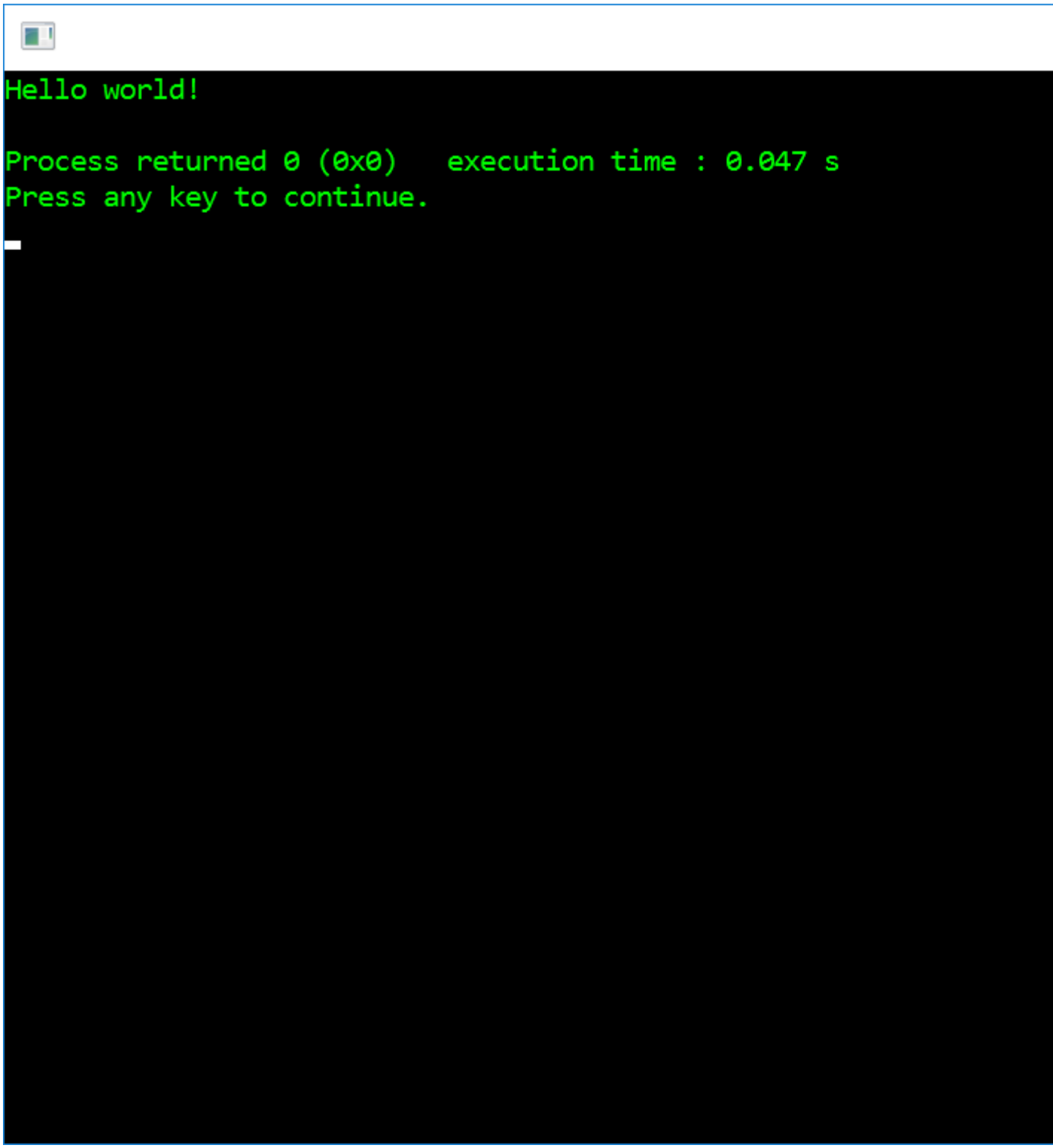

main.cpp [df] - Code::Blocks 16.01
File Edit View Search Project Build Debug Fortran wxSmith Tools Tools+ Plug
Debug
Management
Projects Symbols
Workspace
df
Sources
main.cpp
main.cpp
1 #include <iostream>
2
3 using namespace std;
4
5 int main()
6 {
7 cout << "Hello world!" << en
8 return 0;
9 }
10
Logs & others
Code::Blocks
Search results
Cccc
C:\Users\glind\Desktop\df\main.cpp

Um zu kompilieren, ohne zu laufen, drücken Sie  Um zu starten, ohne erneut zu kompilieren, drücken Sie und zum Kompilieren und Ausführen, drücken Sie .

Um zu starten, ohne erneut zu kompilieren, drücken Sie und zum Kompilieren und Ausführen, drücken Sie .

Um zu starten, ohne erneut zu kompilieren, drücken Sie  und zum Kompilieren und Ausführen, drücken Sie  .

Kompilieren und Ausführen der Standardeinstellung "Hallo Welt!" Code ergibt folgendes Ergebnis:



Kompilieren und Bauen online lesen: https://riptutorial.com/de/cplusplus/topic/4708/kompilieren-und-bauen

Kapitel 66: Komponententest in C ++

Einführung

Komponententests sind eine Stufe im Softwaretest, die das Verhalten und die Richtigkeit von Codeeinheiten überprüft.

In C ++ beziehen sich "Codeeinheiten" häufig auf Klassen, Funktionen oder Gruppen von beiden. Unit-Tests werden häufig mit speziellen "Testing Frameworks" oder "Testing Libraries" durchgeführt, die häufig nicht triviale Syntax- oder Nutzungsmuster verwenden.

In diesem Thema werden verschiedene Strategien und Unit-Testbibliotheken oder -Frameworks beschrieben.

Examples

Google Test

Google Test ist ein von Google gepflegtes C ++ - Testframework. Beim Erstellen einer Testfalldatei müssen Sie die `gtest` Bibliothek `gtest` und mit Ihrem `gtest` verknüpfen.

Minimales Beispiel

```
// main.cpp

#include <gtest/gtest.h>
#include <iostream>

// Google Test test cases are created using a C++ preprocessor macro
// Here, a "test suite" name and a specific "test name" are provided.
TEST(module_name, test_name) {
    std::cout << "Hello world!" << std::endl;
    // Google Test will also provide macros for assertions.
    ASSERT_EQ(1+1, 2);
}

// Google Test can be run manually from the main() function
// or, it can be linked to the gtest_main library for an already
// set-up main() function primed to accept Google Test test cases.
int main(int argc, char** argv) {
    ::testing::InitGoogleTest(&argc, argv);

    return RUN_ALL_TESTS();
}

// Build command: g++ main.cpp -lgtest
```

Fang

Catch ist eine reine Kopfbibliothek, in der Sie sowohl den TDD- als auch den BDD- Geräteteststil verwenden können.

Das folgende Snippet befindet sich auf der Catch-Dokumentationsseite unter diesem Link :

```
SCENARIO( "vectors can be sized and resized", "[vector]" ) {
    GIVEN( "A vector with some items" ) {
        std::vector v( 5 );

        REQUIRE( v.size() == 5 );
        REQUIRE( v.capacity() >= 5 );

        WHEN( "the size is increased" ) {
            v.resize( 10 );

            THEN( "the size and capacity change" ) {
                REQUIRE( v.size() == 10 );
                REQUIRE( v.capacity() >= 10 );
            }
        }
        WHEN( "the size is reduced" ) {
            v.resize( 0 );

            THEN( "the size changes but not capacity" ) {
                REQUIRE( v.size() == 0 );
                REQUIRE( v.capacity() >= 5 );
            }
        }
        WHEN( "more capacity is reserved" ) {
            v.reserve( 10 );

            THEN( "the capacity changes but not the size" ) {
                REQUIRE( v.size() == 5 );
                REQUIRE( v.capacity() >= 10 );
            }
        }
        WHEN( "less capacity is reserved" ) {
            v.reserve( 0 );

            THEN( "neither size nor capacity are changed" ) {
                REQUIRE( v.size() == 5 );
                REQUIRE( v.capacity() >= 5 );
            }
        }
    }
}
```

Praktischerweise werden diese Tests beim Ausführen wie folgt angegeben:

```
Scenario: vectors can be sized and resized
     Given: A vector with some items
      When: more capacity is reserved
      Then: the capacity changes but not the size
```

Komponententest in C ++ online lesen:
https://riptutorial.com/de/cplusplus/topic/9928/komponententest-in-c-plusplus

Kapitel 67: Konstante Klassenmitgliederfunktionen

Bemerkungen

Was bedeutet "const-Memberfunktionen" einer Klasse wirklich? Die einfache Definition scheint zu sein, dass eine const-Member-Funktion das Objekt nicht ändern kann. Was aber "kann sich nicht ändern" bedeutet hier wirklich. Es bedeutet einfach, dass Sie keine Zuordnung für Klassendatenelemente vornehmen können.

Sie können jedoch auch andere indirekte Vorgänge ausführen, beispielsweise das Einfügen eines Eintrags in eine Karte (siehe Beispiel). Wenn Sie dies zulassen, könnte dies so aussehen, als würde die const-Funktion das Objekt ändern (ja, in gewissem Sinne), aber es ist erlaubt.

Die eigentliche Bedeutung ist also, dass eine const-Member-Funktion keine Zuordnung für die Klassendatenvariablen durchführen kann. Aber es kann andere Dinge tun, wie im Beispiel erklärt.

Examples

konstante Mitgliederfunktion

```
#include <iostream>
#include <map>
#include <string>

using namespace std;

class A {
public:
    map<string, string> * mapOfStrings;
public:
    A() {
        mapOfStrings = new map<string, string>();
    }

    void insertEntry(string const & key, string const & value) const {
        (*mapOfStrings)[key] = value;             // This works? Yes it does.
        delete mapOfStrings;                      // This also works
        mapOfStrings = new map<string, string>(); // This * does * not work
    }

    void refresh() {
        delete mapOfStrings;
        mapOfStrings = new map<string, string>(); // Works as refresh is non const function
    }

    void getEntry(string const & key) const {
        cout << mapOfStrings->at(key);
    }
};
```

```
int main(int argc, char* argv[]) {

    A var;
    var.insertEntry("abc", "abcValue");
    var.getEntry("abc");
    getchar();
    return 0;
}
```

Konstante Klassenmitgliederfunktionen online lesen:
https://riptutorial.com/de/cplusplus/topic/7120/konstante-klassenmitgliederfunktionen

Kapitel 68: Kopieren vs Zuordnung

Syntax

- **Konstruktor kopieren**
- MyClass (const MyClass und andere);
- MyClass (MyClass und andere);
- MyClass (volatile const MyClass und andere);
- MyClass (volatile MyClass und andere);
- **Zuweisungskonstruktor**
- MyClass & operator = (const MyClass & rhs);
- MeineKlasse & operator = (MeineKlasse & rhs);
- MyClass & operator = (MyClass rhs);
- const MyClass & operator = (const MyClass & rhs);
- const MyClass & operator = (MyClass & rhs);
- const MyClass & operator = (MyClass rhs);
- MyClass-Operator = (const MyClass & rhs);
- MyClass-Operator = (MyClass & rhs);
- MyClass-Operator = (MyClass rhs);

Parameter

rhs	Rechte Seite der Gleichheit für Kopier- und Zuweisungskonstruktoren. Zum Beispiel der Zuweisungskonstruktor: MyClass operator = (MyClass & rhs);
Platzhalter	Platzhalter

Bemerkungen

Andere gute Ressourcen für die weitere Forschung:

Was ist der Unterschied zwischen Zuweisungsoperator und Kopierkonstruktor?

Zuweisungsoperator vs. Kopierkonstruktor C ++

GeeksForGeeks

C ++ - Artikel

Examples

Aufgabenverwalter

Der Zuweisungsoperator ist, wenn Sie die Daten durch ein bereits vorhandenes (zuvor initialisiertes) Objekt durch die Daten eines anderen Objekts ersetzen. Nehmen wir das als Beispiel:

```
// Assignment Operator
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Foo
{
  public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo(){};
    Foo& operator=(const Foo& rhs)
    {
            data = rhs.data;
            return *this;
    }

    int data;
};

int main()
{
   Foo foo(2); //Foo(int data) called
   Foo foo2(42);
   foo = foo2; // Assignment Operator Called
   cout << foo.data << endl; //Prints 42
}
```

Sie sehen hier, dass ich den Zuweisungsoperator anrufe, wenn ich das `foo` Objekt bereits initialisiert habe. Später dann ordne ich `foo2` `foo` . Alle Änderungen, die angezeigt werden sollen, wenn Sie den Gleichheitszeichenoperator aufrufen, sind in der Funktion `operator=` definiert. Eine ausführbare Ausgabe finden Sie hier: http://cpp.sh/3qtbm

Konstruktor kopieren

Copy-Konstruktor hingegen ist das komplette Gegenteil des Zuweisungskonstruktors. Dieses Mal wird es verwendet, um ein bereits nicht vorhandenes (oder nicht zuvor initialisiertes) Objekt zu initialisieren. Das bedeutet, dass alle Daten des Objekts, dem Sie es zuweisen, kopiert werden, ohne dass das Objekt, auf das kopiert wird, tatsächlich initialisiert wird. Sehen wir uns nun den gleichen Code wie zuvor an, ändern Sie jedoch den Zuweisungskonstruktor als Kopierkonstruktor:

```
// Copy Constructor
#include <iostream>
#include <string>

using std::cout;
using std::endl;
```

```
class Foo
{
  public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo(){};
    Foo(const Foo& rhs)
    {
            data = rhs.data;
    }

    int data;
};

int main()
{
   Foo foo(2); //Foo(int data) called
   Foo foo2 = foo; // Copy Constructor called
   cout << foo2.data << endl;
}
```

Sie können hier sehen `Foo foo2 = foo;` In der Hauptfunktion weise ich das Objekt sofort zu, bevor es tatsächlich initialisiert wird. Dies bedeutet, wie bereits gesagt, ein Kopierkonstruktor. `foo2` , dass ich den Parameter int nicht für das Objekt `foo2` , da ich automatisch die vorherigen Daten aus dem Objekt foo `foo2` . Hier ist eine Beispielausgabe: http://cpp.sh/5iu7

Copy Constructor Vs Zuweisungskonstruktor

Ok, wir haben kurz nachgeschaut, was der Kopierkonstruktor und der Zuweisungskonstruktor oben sind, und es wurden jeweils Beispiele gegeben. Lassen Sie uns beide im gleichen Code sehen. Dieser Code ist ähnlich wie oben. Nehmen wir das an:

```
// Copy vs Assignment Constructor
#include <iostream>
#include <string>

using std::cout;
using std::endl;

class Foo
{
  public:
    Foo(int data)
    {
        this->data = data;
    }
    ~Foo(){};
    Foo(const Foo& rhs)
    {
            data = rhs.data;
    }

    Foo& operator=(const Foo& rhs)
    {
```

```
        data = rhs.data;
        return *this;
    }

    int data;
};

int main()
{
   Foo foo(2); //Foo(int data) / Normal Constructor called
   Foo foo2 = foo; //Copy Constructor Called
   cout << foo2.data << endl;

   Foo foo3(42);
   foo3=foo; //Assignment Constructor Called
   cout << foo3.data << endl;
}
```

Ausgabe:

```
2
2
```

Hier sehen Sie, dass wir zuerst den Copy-Konstruktor aufrufen, indem Sie die Zeile `Foo foo2 = foo;` ausführen `Foo foo2 = foo;` . Da wir es vorher nicht initialisiert haben. Und dann rufen wir als Nächstes den Zuweisungsoperator auf foo3 auf, da er bereits initialisiert wurde `foo3=foo` ;

Kopieren vs Zuordnung online lesen: https://riptutorial.com/de/cplusplus/topic/7158/kopieren-vs-zuordnung

Kapitel 69: Kopplungsspezifikationen

Einführung

Eine Verknüpfungsspezifikation weist den Compiler an, Deklarationen so zu kompilieren, dass sie mit Deklarationen verknüpft werden können, die in einer anderen Sprache wie C geschrieben sind.

Syntax

- externes *String-Literal* { *Deklaration-Seq* (*Opt*)}
- externe *String-Literal- Deklaration*

Bemerkungen

Der Standard verlangt, dass alle Compiler `extern "C"` , damit C ++ mit C und `extern "C++"` kompatibel ist, was dazu verwendet werden kann, eine umschließende Verbindungsspezifikation zu überschreiben und den Standard wiederherzustellen. Andere unterstützte Verbindungsspezifikationen sind implementierungsdefiniert .

Examples

Signalhandler für ein Unix-ähnliches Betriebssystem

Da vom Kernel ein Signalhandler mit der C-Aufrufkonvention aufgerufen wird, muss der Compiler beim Compilieren der Funktion aufgefordert werden, die C-Aufrufkonvention zu verwenden.

```
volatile sig_atomic_t death_signal = 0;
extern "C" void cleanup(int signum) {
    death_signal = signum;
}
int main() {
    bind(...);
    listen(...);
    signal(SIGTERM, cleanup);
    while (int fd = accept(...)) {
        if (fd == -1 && errno == EINTR && death_signal) {
            printf("Caught signal %d; shutting down\n", death_signal);
            break;
        }
        // ...
    }
}
```

Einen C-Bibliotheksheader mit C ++ kompatibel machen

AC-Bibliotheksheader können normalerweise in ein C ++ - Programm eingefügt werden, da die meisten Deklarationen sowohl in C als auch in C ++ gültig sind. Betrachten Sie zum Beispiel

folgendes `foo.h` :

```
typedef struct Foo {
    int bar;
} Foo;
Foo make_foo(int);
```

Die Definition von `make_foo` wird separat zusammengestellt und mit dem Header in Objektform verteilt.

Ein C ++ - Programm kann `#include <foo.h>` , der Compiler weiß jedoch nicht, dass die Funktion `make_foo` als C-Symbol definiert ist, und versucht wahrscheinlich, es mit einem entstellten Namen zu suchen und kann es nicht finden. Selbst wenn die Definition von `make_foo` in der Bibliothek gefunden werden kann, verwenden nicht alle Plattformen die gleichen Aufrufkonventionen für C und C ++, und der C ++ - Compiler verwendet beim Aufruf von `make_foo` die C ++ - Aufrufkonvention, die bei `make_foo` wahrscheinlich zu einem Segmentierungsfehler `make_foo` erwartet, mit der C-Aufrufkonvention aufgerufen zu werden.

Um dieses Problem zu beheben, können Sie fast alle Deklarationen im Header in einen `extern "C"` -Block `extern "C"` .

```
#ifdef __cplusplus
extern "C" {
#endif

typedef struct Foo {
    int bar;
} Foo;
Foo make_foo(int);

#ifdef __cplusplus
}  /* end of "extern C" block */
#endif
```

Wenn nun `foo.h` aus einem C-Programm enthalten ist, wird es nur als gewöhnliche Deklaration `foo.h` Wenn jedoch `foo.h` aus einem C ++ - Programm enthalten ist, befindet sich `make_foo` in einem `extern "C"` -Block, und der Compiler weiß zu suchen einen unveränderten Namen und verwenden Sie die C-Aufrufkonvention.

Kopplungsspezifikationen online lesen:
https://riptutorial.com/de/cplusplus/topic/9268/kopplungsspezifikationen

Kapitel 70: Lambdas

Syntax

- [*default-capture* , *capture-list*] (*Argumentliste*) veränderliche *Attribute für Throw-Spezifikation -> Rückgabetyp* { *lambda-body* } // Reihenfolge der Lambda-Bezeichner und Attribute
- [*Erfassungsliste*] (*Argumentliste*) { *Lambda-Body* } // Allgemeine Lambda-Definition.
- [=] (*argument-list*) { *lambda-body* } // Erfasst alle benötigten lokalen Variablen nach Wert.
- [&] (*argument-list*) { *lambda-body* } // Erfasst alle benötigten lokalen Variablen per Verweis.
- [*capture-list*] { *lambda-body* } // Argumentliste und Angaben können weggelassen werden.

Parameter

Parameter	Einzelheiten
Standard-Capture	Gibt an, wie alle nicht aufgelisteten Variablen erfasst werden. Kann `=` (Erfassung nach Wert) oder `&` (Aufnahme nach Referenz) sein. Ohne Angabe sind nicht aufgeführte Variablen im *Lambda-Body nicht erreichbar* . Das *Standard-Capture* muss vor der *Capture-Liste stehen* .
Capture-Liste	Gibt an, wie lokale Variablen im *Lambda-Body verfügbar gemacht werden* . Variablen ohne Präfix werden nach Wert erfasst. Mit `&` Variablen werden durch Verweis erfasst. Innerhalb einer Klassenmethode, `this` kann verwendet werden , um alle zugänglich ihre Mitglieder durch Bezugnahme zu machen. Auf nicht aufgelistete Variablen kann nicht zugegriffen werden, es sei denn, der Liste ist eine *Standarderfassung* vorangestellt.
Argumentliste	Gibt die Argumente der Lambda-Funktion an.
veränderlich	*(optional)* Normalerweise werden durch Wert erfasste Variablen `const` . Durch die Angabe der `mutable` Eigenschaft werden sie nicht konstant. Änderungen an diesen Variablen werden zwischen Aufrufen beibehalten.
Wurfspezifikation	*(optional)* Gibt das Auslöseverhalten der Lambda-Funktion an. Zum Beispiel: `noexcept` oder `throw(std::exception)` .
Attribute	*(optional)* Beliebige Attribute für die Lambda-Funktion. Wenn zum Beispiel der *Lambda-Body* immer eine Ausnahme `[[noreturn]]` kann `[[noreturn]]` verwendet werden.
-> Rückgabetyp	*(optional)* Gibt den Rückgabetyp der Lambda-Funktion an. Erforderlich, wenn der Rückgabetyp nicht vom Compiler bestimmt werden kann.
Lambda-Körper	Ein Codeblock, der die Implementierung der Lambda-Funktion enthält.

Bemerkungen

C ++ 17 (der aktuelle Entwurf) führt `constexpr` Lambdas ein, im Wesentlichen Lambdas, die zur Kompilierzeit ausgewertet werden können. Ein Lambda ist automatisch `constexpr` wenn es die Anforderungen von `constexpr` erfüllt. Sie können es jedoch auch mit dem Schlüsselwort `constexpr` angeben:

```
//Explicitly define this lambdas as constexpr
[]() constexpr {
    //Do stuff
}
```

Examples

Was ist ein Lambda-Ausdruck?

Mit einem **Lambda-Ausdruck können Sie** einfache Funktionsobjekte auf einfache Weise erstellen. Ein Lambda-Ausdruck ist ein Prvalue, dessen Ergebnisobjekt Schließobjekt genannt wird , das sich wie ein Funktionsobjekt verhält.

Der Name "Lambda-Ausdruck" stammt von Lambda-Kalkül , einem mathematischen Formalismus, der in den 1930er Jahren von Alonzo Church zur Untersuchung von Fragen zu Logik und Berechenbarkeit erfunden wurde. Lambda-Kalkül bildete die Grundlage von LISP , einer funktionalen Programmiersprache. Im Vergleich zu Lambda-Kalkül und LISP weisen C ++ - Lambda-Ausdrücke die Eigenschaften auf, unbenannt zu sein und Variablen aus dem umgebenden Kontext zu erfassen, ihnen fehlt jedoch die Fähigkeit, Funktionen auszuführen und zurückzugeben.

Ein Lambda-Ausdruck wird häufig als Argument für Funktionen verwendet, die ein aufrufbares Objekt enthalten. Dies kann einfacher sein als das Erstellen einer benannten Funktion, die nur verwendet wird, wenn sie als Argument übergeben wird. In solchen Fällen werden im Allgemeinen Lambda-Ausdrücke bevorzugt, da sie es ermöglichen, die Funktionsobjekte inline zu definieren.

Ein Lambda besteht normalerweise aus drei Teilen: einer Erfassungsliste `[]` , einer optionalen Parameterliste `()` und einem Hauptteil `{}` , die alle leer sein können:

```
[](){}                // An empty lambda, which does and returns nothing
```

Aufnahmeliste

`[]` ist die **Erfassungsliste** . Auf Variablen des umschließenden Bereichs kann standardmäßig kein Lambda zugreifen. *Durch das Erfassen* einer Variablen ist sie innerhalb des Lambda als Kopie oder als Referenz verfügbar . Die erfassten Variablen werden Teil des Lambda. Im Gegensatz zu Funktionsargumenten müssen sie beim Aufruf des Lambda nicht übergeben werden.

```
int a = 0;                        // Define an integer variable
```

```
auto f = []()   { return a*9; }; // Error: 'a' cannot be accessed
auto f = [a]()  { return a*9; }; // OK, 'a' is "captured" by value
auto f = [&a]() { return a++; }; // OK, 'a' is "captured" by reference
                                 //      Note: It is the responsibility of the programmer
                                 //      to ensure that a is not destroyed before the
                                 //      lambda is called.
auto b = f();                    // Call the lambda function. a is taken from the capture list
and not passed here.
```

Parameterliste

`()` ist die **Parameterliste** , die fast die gleiche ist wie bei regulären Funktionen. Wenn das Lambda keine Argumente akzeptiert, können diese Klammern weggelassen werden (es sei denn, Sie müssen das Lambda als `mutable` deklarieren). Diese beiden Lambdas sind gleichwertig:

```
auto call_foo  = [x](){ x.foo(); };
auto call_foo2 = [x]{ x.foo(); };
```

C ++ 14

Die Parameterliste kann anstelle von tatsächlichen Typen den Platzhaltertyp `auto` verwenden. Dadurch verhält sich dieses Argument wie ein Vorlagenparameter einer Funktionsvorlage. Die folgenden Lambdas sind gleichwertig, wenn Sie einen Vektor in generischem Code sortieren möchten:

```
auto sort_cpp11 = [](std::vector<T>::const_reference lhs, std::vector<T>::const_reference rhs)
{ return lhs < rhs; };
auto sort_cpp14 = [](const auto &lhs, const auto &rhs) { return lhs < rhs; };
```

Funktionskörper

`{}` ist der **Körper** , der dem von regulären Funktionen entspricht.

Ein Lambda anrufen

Das Ergebnisobjekt eines Lambda-Ausdrucks ist eine Schließung , die mit dem `operator()` aufgerufen werden kann (wie bei anderen Funktionsobjekten):

```
int multiplier = 5;
auto timesFive = [multiplier](int a) { return a * multiplier; };
std::out << timesFive(2); // Prints 10

multiplier = 15;
std::out << timesFive(2); // Still prints 2*5 == 10
```

Rückgabetyp

Standardmäßig wird der Rückgabetyp eines Lambda-Ausdrucks abgeleitet.

```
[](){ return true; };
```

In diesem Fall ist der Rückgabetyp `bool` .

Sie können den Rückgabetyp auch manuell mit der folgenden Syntax angeben:

```
[]() -> bool { return true; };
```

Mutable Lambda

Objekte, die im Lambda-Wert erfasst werden, sind standardmäßig unveränderlich. Dies liegt daran, dass der `operator()` des generierten Abschlussobjekts standardmäßig `const` ist.

```
auto func = [c = 0](){++c; std::cout << c;};  // fails to compile because ++c
                                              // tries to mutate the state of
                                              // the lambda.
```

Das Ändern kann mit dem Schlüsselwort `mutable` erlaubt werden, wodurch der `operator()` des näheren Objekts nicht- `const` :

```
auto func = [c = 0]() mutable {++c; std::cout << c;};
```

Bei Verwendung zusammen mit dem Rückgabetyp wird das `mutable` davor `mutable` .

```
auto func = [c = 0]() mutable -> int {++c; std::cout << c; return c;};
```

Ein Beispiel zur Veranschaulichung der Nützlichkeit von Lambdas

Vor C ++ 11:

C ++ 11

```
// Generic functor used for comparison
struct islessthan
{
    islessthan(int threshold) : _threshold(threshold) {}

    bool operator()(int value) const
    {
        return value < _threshold;
    }
private:
    int _threshold;
};

// Declare a vector
const int arr[] = { 1, 2, 3, 4, 5 };
std::vector<int> vec(arr, arr+5);

// Find a number that's less than a given input (assume this would have been function input)
int threshold = 10;
std::vector<int>::iterator it = std::find_if(vec.begin(), vec.end(), islessthan(threshold));
```

Seit C ++ 11:

```
// Declare a vector
std::vector<int> vec{ 1, 2, 3, 4, 5 };

// Find a number that's less than a given input (assume this would have been function input)
int threshold = 10;
auto it = std::find_if(vec.begin(), vec.end(), [threshold](int value) { return value <
threshold; });
```

Angabe des Rückgabetyps

Bei Lambdas mit einer einzelnen return-Anweisung oder mehreren return-Anweisungen, deren Ausdrücke denselben Typ haben, kann der Compiler den Rückgabetyp ableiten:

```
// Returns bool, because "value > 10" is a comparison which yields a Boolean result
auto l = [](int value) {
    return value > 10;
}
```

Bei Lambdas mit mehreren return-Anweisungen *verschiedener* Typen kann der Compiler den Rückgabetyp nicht ableiten:

```
// error: return types must match if lambda has unspecified return type
auto l = [](int value) {
    if (value < 10) {
        return 1;
    } else {
        return 1.5;
    }
};
```

In diesem Fall müssen Sie den Rückgabetyp explizit angeben:

```
// The return type is specified explicitly as 'double'
auto l = [](int value) -> double {
    if (value < 10) {
        return 1;
    } else {
        return 1.5;
    }
};
```

Die Regeln hierfür entsprechen den Regeln für die `auto` Typabzug. Lambdas ohne explizit angegebene Rückgabetypen geben niemals Referenzen zurück. Wenn ein Referenztyp gewünscht wird, muss er auch explizit angegeben werden:

```
auto copy = [](X& x) { return x; };       // 'copy' returns an X, so copies its input
auto ref  = [](X& x) -> X& { return x; }; // 'ref' returns an X&, no copy
```

Erfassung nach Wert

Wenn Sie den Namen der Variablen in der Erfassungsliste angeben, erfasst der Lambda ihn nach

Wert. Dies bedeutet, dass der generierte Abschlusstyp für das Lambda eine Kopie der Variablen speichert. Dies erfordert auch , dass die Art Variable sein *copy-konstruierbar:*

```
int a = 0;

[a]() {
    return a;   // Ok, 'a' is captured by value
};
```

C ++ 14

```
auto p = std::unique_ptr<T>(...);

[p]() {         // Compile error; `unique_ptr` is not copy-constructible
    return p->createWidget();
};
```

Ab C ++ 14 können Variablen vor Ort initialisiert werden. Auf diese Weise können nur Arten von Bewegungen im Lambda erfasst werden.

C ++ 14

```
auto p = std::make_unique<T>(...);

[p = std::move(p)]() {
    return p->createWidget();
};
```

Obwohl ein Lambda Variablen nach Wert erfasst, wenn sie durch ihren Namen angegeben werden, können diese Variablen nicht standardmäßig innerhalb des Lambda-Körpers geändert werden. Dies liegt daran, dass der Schließungstyp den Lambda-Body in eine Deklaration von `operator() const` .

Die `const` gilt für Zugriffe auf Membervariablen des Abschlusstyps und erfasste Variablen, die Mitglieder des Closures sind (alle gegenteiligen Anschein)

```
int a = 0;

[a]() {
    a = 2;      // Illegal, 'a' is accessed via `const`

    decltype(a) a1 = 1;
    a1 = 2; // valid: variable 'a1' is not const
};
```

Um das `const` zu entfernen, müssen Sie das Schlüsselwort `mutable` im Lambda angeben:

```
int a = 0;

[a]() mutable {
    a = 2;      // OK, 'a' can be modified
    return a;
};
```

Da `a` durch Wert erfasst wurde, wirken sich Änderungen durch Aufruf des Lambda nicht auf `a` . Der Wert von `a` wurde in den Lambda kopiert , wenn es gebaut wurde, so die Kopie des Lambda `a` ist unabhängig von der externen `a` Variable.

```
int a = 5 ;
auto plus5Val = [a] (void) { return a + 5 ; } ;
auto plus5Ref = [&a] (void) {return a + 5 ; } ;

a = 7 ;
std::cout << a << ", value " << plus5Val() << ", reference " << plus5Ref() ;
// The result will be "7, value 10, reference 12"
```

Generalisierte Erfassung

C ++ 14

Lambdas können Ausdrücke und nicht nur Variablen erfassen. Auf diese Weise können Lambdas Nur-Bewegung-Typen speichern:

```
auto p = std::make_unique<T>(...);

auto lamb = [p = std::move(p)]() //Overrides capture-by-value of `p`.
{
  p->SomeFunc();
};
```

Dadurch wird die äußere `p` Variable in die Lambda-Capture-Variable, auch als `p` bezeichnet, `p` . `lamb` besitzt nun den von `make_unique` zugewiesenen `make_unique` . Da der Abschluss einen Typ enthält, der nicht kopierbar ist, bedeutet dies, dass `lamb` selbst nicht kopierbar ist. Es kann aber verschoben werden:

```
auto lamb_copy = lamb; //Illegal
auto lamb_move = std::move(lamb); //legal.
```

Jetzt besitzt `lamb_move` die Erinnerung.

Beachten Sie, dass für `std::function<>` die gespeicherten Werte kopierbar sein müssen. Sie können Ihre eigene move-only-erfordernde `std::function` schreiben oder das Lambda einfach in einen `shared_ptr` Wrapper `shared_ptr` :

```
auto shared_lambda = [](auto&& f){
  return [spf = std::make_shared<std::decay_t<decltype(f)>>(decltype(f)(f))]
  (auto&&...args)->decltype(auto) {
    return (*spf)(decltype(args)(args)...);
  };
};
auto lamb_shared = shared_lambda(std::move(lamb_move));
```

nimmt unseren Einzug nur Lambda und stopft seinen Zustand in einen gemeinsamen Zeiger gibt dann eine Lambda, die kopiert werden und dann in einer gespeicherten `std::function` oder

ähnlichem.

Die generalisierte Erfassung verwendet den `auto` Typabzug für den Variablentyp. Diese Captures werden standardmäßig als Werte deklariert, sie können jedoch auch Referenzen sein:

```
int a = 0;

auto lamb = [&v = a](int add) //Note that `a` and `v` have different names
{
  v += add; //Modifies `a`
};

lamb(20); //`a` becomes 20.
```

Die Generalize-Erfassung muss überhaupt keine externe Variable erfassen. Es kann einen beliebigen Ausdruck erfassen:

```
auto lamb = [p = std::make_unique<T>(...)]()
{
    p->SomeFunc();
}
```

Dies ist nützlich, um Lambdas beliebige Werte zu geben, die sie speichern und möglicherweise modifizieren können, ohne sie extern zum Lambda deklarieren zu müssen. Das ist natürlich nur sinnvoll, wenn Sie nicht auf diese Variablen zugreifen möchten, nachdem das Lambda seine Arbeit beendet hat.

Nach Referenz erfassen

Wenn Sie dem Namen einer lokalen Variable ein `&` voranstellen, wird die Variable als Referenz erfasst. Konzeptionell bedeutet dies, dass der Schließungstyp des Lambda eine Referenzvariable hat, die als Referenz auf die entsprechende Variable von außerhalb des Gültigkeitsbereichs des Lambda initialisiert wird. Jede Verwendung der Variablen im Lambda-Body bezieht sich auf die ursprüngliche Variable:

```
// Declare variable 'a'
int a = 0;

// Declare a lambda which captures 'a' by reference
auto set = [&a]() {
    a = 1;
};

set();
assert(a == 1);
```

Das Schlüsselwort `mutable` wird nicht benötigt, da `a` selbst nicht `const` .

Erfassen durch Referenz bedeutet natürlich, dass das Lambda **nicht** dem Umfang der erfassten Variablen entgehen **darf** . Sie können also Funktionen aufrufen, die eine Funktion übernehmen, aber Sie dürfen keine Funktion aufrufen, die das Lambda über den Bereich Ihrer Referenzen

hinaus *speichert* . Und du darfst das Lambda nicht zurückgeben.

Standarderfassung

Auf lokale Variablen, die nicht explizit in der Erfassungsliste angegeben sind, kann standardmäßig nicht vom Lambda-Body aus zugegriffen werden. Es ist jedoch möglich, durch den Lambda-Body benannte Variablen implizit zu erfassen:

```
int a = 1;
int b = 2;

// Default capture by value
[=]() { return a + b; }; // OK; a and b are captured by value

// Default capture by reference
[&]() { return a + b; }; // OK; a and b are captured by reference
```

Die explizite Erfassung kann immer noch neben der impliziten Standarderfassung erfolgen. Die explizite Erfassungsdefinition überschreibt die Standarderfassung:

```
int a = 0;
int b = 1;

[=, &b]() {
    a = 2; // Illegal; 'a' is capture by value, and lambda is not 'mutable'
    b = 2; // OK; 'b' is captured by reference
};
```

Generische Lambdas

c ++ 14

Lambda-Funktionen können Argumente beliebigen Typs annehmen. Dadurch kann ein Lambda generischer sein:

```
auto twice = [](auto x){ return x+x; };

int i = twice(2); // i == 4
std::string s = twice("hello"); // s == "hellohello"
```

Dies wird in C ++ implementiert, indem der `operator()` des Abschlusstyps eine Vorlagenfunktion wird. Der folgende Typ hat das gleiche Verhalten wie der obige Lambda-Verschluss:

```
struct _unique_lambda_type
{
  template<typename T>
  auto operator() (T x) const {return x + x;}
};
```

Nicht alle Parameter in einem generischen Lambda müssen generisch sein:

```
[](auto x, int y) {return x + y;}
```

Hier wird `x` basierend auf dem ersten Funktionsargument abgeleitet, während `y` immer `int` .

Generische Lambdas können auch Argumente als Referenz verwenden, wobei die üblichen Regeln für `auto` und `&` . Wenn ein generischer Parameter als `auto&&` , ist dies eine *Weiterleitungsreferenz* auf das übergebene Argument und keine *rvalue-* Referenz :

```
auto lamb1 = [](int &&x) {return x + 5;};
auto lamb2 = [](auto &&x) {return x + 5;};
int x = 10;
lamb1(x); // Illegal; must use `std::move(x)` for `int&&` parameters.
lamb2(x); // Legal; the type of `x` is deduced as `int&`.
```

Lambda-Funktionen können variadisch sein und ihre Argumente perfekt weiterleiten:

```
auto lam = [](auto&&... args){return f(std::forward<decltype(args)>(args)...);};
```

oder:

```
auto lam = [](auto&&... args){return f(decltype(args)(args)...);};
```

was nur "richtig" mit Variablen vom Typ `auto&&` funktioniert.

Ein guter Grund, generische Lambdas zu verwenden, ist der Besuch der Syntax.

```
boost::variant<int, double> value;
apply_visitor(value, [&](auto&& e){
  std::cout << e;
});
```

Hier besuchen wir polymorph. In anderen Zusammenhängen sind die Namen des Typs, den wir übergeben, nicht interessant:

```
mutex_wrapped<std::ostream&> os = std::cout;
os.write([&](auto&& os){
  os << "hello world\n";
});
```

Das Wiederholen des Typs von `std::ostream&` ist hier Rauschen. Es wäre so, als müssten Sie bei jeder Verwendung den Typ einer Variablen erwähnen. Hier schaffen wir einen Besucher, aber keinen polymorphen Besucher; `auto` wird aus dem gleichen Grund verwendet, in dem Sie `auto` in einer `for(:)` Schleife verwenden können.

Konvertierung in Funktionszeiger

Wenn die Capture-Liste eines Lambdas leer ist, wird das Lambda implizit in einen Funktionszeiger konvertiert, der dieselben Argumente verwendet und denselben Rückgabetyp zurückgibt:

```
auto sorter = [](int lhs, int rhs) -> bool {return lhs < rhs;};

using func_ptr = bool(*)(int, int);
func_ptr sorter_func = sorter; // implicit conversion
```

Eine solche Konvertierung kann auch mit einem unären Plus-Operator durchgeführt werden:

```
func_ptr sorter_func2 = +sorter; // enforce implicit conversion
```

Das Aufrufen dieses Funktionszeigers verhält sich genauso wie das Aufrufen von `operator()` auf dem Lambda. Dieser Funktionszeiger ist in keiner Weise von der Existenz des Quell-Lambda-Verschlusses abhängig. Sie kann daher den Lambda-Verschluss überleben.

Diese Funktion ist vor allem für die Verwendung von Lambdas mit APIs hilfreich, die Funktionszeiger anstelle von C ++ - Funktionsobjekten verwenden.

C ++ 14

Die Konvertierung in einen Funktionszeiger ist auch für generische Lambdas mit leerer Erfassungsliste möglich. Bei Bedarf wird ein Vorlagenargumentenabzug verwendet, um die richtige Spezialisierung auszuwählen.

```
auto sorter = [](auto lhs, auto rhs) { return lhs < rhs; };
using func_ptr = bool(*)(int, int);
func_ptr sorter_func = sorter;  // deduces int, int
// note however that the following is ambiguous
// func_ptr sorter_func2 = +sorter;
```

Klasse Lambdas und Capture davon

Ein in einer Member-Funktion einer Klasse ausgewerteter Lambda-Ausdruck ist implizit ein Freund dieser Klasse:

```
class Foo
{
private:
    int i;

public:
    Foo(int val) : i(val) {}

    // definition of a member function
    void Test()
    {
        auto lamb = [](Foo &foo, int val)
        {
            // modification of a private member variable
            foo.i = val;
        };

        // lamb is allowed to access a private member, because it is a friend of Foo
        lamb(*this, 30);
    }
```

```
};
```

Ein solches Lambda ist nicht nur ein Freund dieser Klasse, es hat den gleichen Zugriff wie die Klasse, in der es deklariert ist.

Lambdas können `this` Zeiger erfassen, der die Objektinstanz darstellt, für die die äußere Funktion aufgerufen wurde. Dazu fügen Sie `this` der Capture-Liste hinzu:

```
class Foo
{
private:
    int i;

public:
    Foo(int val) : i(val) {}

    void Test()
    {
        // capture the this pointer by value
        auto lamb = [this](int val)
        {
            i = val;
        };

        lamb(30);
    }
};
```

Wenn `this` erfasst wird, kann das Lambda die Namen der enthaltenen Klasse so verwenden, als wäre es in der enthaltenen Klasse. Ein implizites `this->` wird also auf solche Mitglieder angewendet.

Beachten Sie, dass `this` vom Wert erfasst wird, nicht jedoch vom Wert des Typs. Es wird durch den Wert von `this` erfasst, der ein *Zeiger ist* . Als solches ist das Lambda nicht *besitzt* `this` . Wenn das Lambda-Out die Lebensdauer des Objekts, das es erstellt hat, lebt, kann das Lambda ungültig werden.

Das bedeutet auch , dass das Lambda ändern kann `this` ohne erklärt zu werden `mutable` . Es ist der Zeiger, der `const` , nicht das Objekt, auf das gezeigt wird. Das heißt, es sei denn, die äußere Elementfunktion war selbst eine `const` Funktion.

Beachten Sie außerdem, dass die Standard - Capture - Klauseln, die beide `[=]` und `[&]` , wird *auch* erfassen `this` implizit. Und beide erfassen es durch den Wert des Zeigers. Es ist in der Tat ein Fehler, `this` in der Capture-Liste anzugeben, wenn ein Standardwert angegeben wird.

C ++ 17

Lambda-Dateien können eine Kopie `this` Objekts erfassen, das zum Zeitpunkt der Lambda-Erstellung erstellt wurde. Dies geschieht durch Hinzufügen von `*this` zur Aufnahmeliste:

```
class Foo
{
private:
```

```
    int i;

public:
    Foo(int val) : i(val) {}

    void Test()
    {
        // capture a copy of the object given by the this pointer
        auto lamb = [*this](int val) mutable
        {
            i = val;
        };

        lamb(30); // does not change this->i
    }
};
```

Übertragen von Lambda-Funktionen nach C ++ 03 mithilfe von Funktoren

Lambda-Funktionen in C ++ sind syntaktischer Zucker, der eine sehr kurze Syntax für das Schreiben von Funktoren bietet. Daher kann eine äquivalente Funktionalität in C ++ 03 erhalten werden (wenn auch viel ausführlicher), indem die Lambda-Funktion in einen Funktor konvertiert wird:

```
// Some dummy types:
struct T1 {int dummy;};
struct T2 {int dummy;};
struct R {int dummy;};

// Code using a lambda function (requires C++11)
R use_lambda(T1 val, T2 ref) {
  // Use auto because the type of the lambda is unknown.
  auto lambda = [val, &ref](int arg1, int arg2) -> R {
    /* lambda-body */
    return R();
  };
  return lambda(12, 27);
}

// The functor class (valid C++03)
// Similar to what the compiler generates for the lambda function.
class Functor {
  // Capture list.
  T1 val;
  T2& ref;

public:
  // Constructor
  inline Functor(T1 val, T2& ref) : val(val), ref(ref) {}

  // Functor body
  R operator()(int arg1, int arg2) const {
    /* lambda-body */
    return R();
  }
};

// Equivalent to use_lambda, but uses a functor (valid C++03).
```

```
R use_functor(T1 val, T2 ref) {
  Functor functor(val, ref);
  return functor(12, 27);
}

// Make this a self-contained example.
int main() {
  T1 t1;
  T2 t2;
  use_functor(t1,t2);
  use_lambda(t1,t2);
  return 0;
}
```

Wenn die Lambda-Funktion `mutable` ist, machen Sie den Aufrufoperator des Funkers zu non-const, dh:

```
R operator()(int arg1, int arg2) /*non-const*/ {
  /* lambda-body */
  return R();
}
```

Rekursive Lambdas

Nehmen wir an, wir möchten Euclids `gcd()` als Lambda schreiben. Als Funktion ist es:

```
int gcd(int a, int b) {
    return b == 0 ? a : gcd(b, a%b);
}
```

Ein Lambda kann jedoch nicht rekursiv sein, es gibt keine Möglichkeit, sich selbst anzurufen. Ein Lambda hat keinen Namen und die Verwendung `this` in den Körper eines Lambda bezieht sich auf ein erfasstes `this` (die Lambda unter der Annahme , wird im Körper einer Elementfunktion geschaffen, sonst ist es ein Fehler). Wie lösen wir dieses Problem?

Verwenden Sie die `std::function`

Wir können eine Lambda-Capture-Referenz zu einer noch nicht erstellten `std::function` :

```
std::function<int(int, int)> gcd = [&](int a, int b){
    return b == 0 ? a : gcd(b, a%b);
};
```

Das funktioniert, sollte aber sparsam eingesetzt werden. Es ist langsam (wir verwenden jetzt Typenlöschung anstelle eines direkten Funktionsaufrufs), es ist fragil (das Kopieren von `gcd` oder das Zurückgeben von `gcd` wird `gcd` , da das Lambda auf das ursprüngliche Objekt verweist), und es funktioniert nicht mit generischen Lambdas.

Mit zwei intelligenten Zeigern:

```
auto gcd_self = std::make_shared<std::unique_ptr< std::function<int(int, int)> >>();
```

```
*gcd_self = std::make_unique<std::function<int(int, int)>>(
  [gcd_self](int a, int b){
    return b == 0 ? a : (**gcd_self)(b, a%b);
  };
};
```

Dies führt zu einer erheblichen Anzahl von Umleitungen (was Overhead ist), aber es kann kopiert / zurückgegeben werden, und alle Kopien teilen sich den Status. Es erlaubt Ihnen, das Lambda zurückzugeben, und ist sonst weniger zerbrechlich als die obige Lösung.

Verwenden Sie einen Y-Kombinator

Mit Hilfe einer kurzen Utility-Struktur können wir all diese Probleme lösen:

```
template <class F>
struct y_combinator {
    F f; // the lambda will be stored here

    // a forwarding operator():
    template <class... Args>
    decltype(auto) operator()(Args&&... args) const {
        // we pass ourselves to f, then the arguments.
        // the lambda should take the first argument as `auto&& recurse` or similar.
        return f(*this, std::forward<Args>(args)...);
    }
};
// helper function that deduces the type of the lambda:
template <class F>
y_combinator<std::decay_t<F>> make_y_combinator(F&& f) {
    return {std::forward<F>(f)};
}
// (Be aware that in C++17 we can do better than a `make_` function)
```

wir können unsere `gcd` wie `gcd` implementieren:

```
auto gcd = make_y_combinator(
  [](auto&& gcd, int a, int b){
    return b == 0 ? a : gcd(b, a%b);
  }
);
```

Der `y_combinator` ist ein Konzept aus dem Lambda-Kalkül, mit dem Sie eine Rekursion haben können, ohne sich selbst benennen zu können, bis Sie definiert sind. Das ist genau das Problem, das Lambdas haben.

Sie erstellen ein Lambda, das als erstes Argument "recurse" verwendet. Wenn Sie rekursieren möchten, übergeben Sie die Argumente, um erneut aufzurufen.

Der `y_combinator` dann ein Funktionsobjekt zurück, das diese Funktion mit seinen Argumenten aufruft, jedoch mit einem geeigneten "recurse" -Objekt (nämlich dem `y_combinator` selbst) als erstes Argument. Die übrigen Argumente, mit denen Sie den `y_combinator` aufrufen, werden `y_combinator` an das Lambda weitergeleitet.

Zusamenfassend:

```
auto foo = make_y_combinator( [&](auto&& recurse, some arguments) {
  // write body that processes some arguments
  // when you want to recurse, call recurse(some other arguments)
});
```

und Sie haben eine Rekursion in einem Lambda ohne gravierende Einschränkungen oder erheblichen Mehraufwand.

Verwenden von Lambdas zum Auspacken des Inline-Parameterpakets

C ++ 14

Das Auspacken von Parametern erfordert normalerweise das Schreiben einer Hilfsfunktion für jedes Mal, wenn Sie dies tun möchten.

In diesem Spielzeugbeispiel:

```
template<std::size_t...Is>
void print_indexes( std::index_sequence<Is...> ) {
  using discard=int[];
  (void)discard{0,((void)(
    std::cout << Is << '\n' // here Is is a compile-time constant.
  ),0)...};
}
template<std::size_t I>
void print_indexes_upto() {
  return print_indexes( std::make_index_sequence<I>{} );
}
```

Der `print_indexes_upto` möchte ein Parameterpaket mit Indizes erstellen und auspacken. Dazu muss eine Hilfsfunktion aufgerufen werden. Jedes Mal, wenn Sie ein von Ihnen erstelltes Parameterpaket auspacken möchten, müssen Sie dazu eine benutzerdefinierte Hilfsfunktion erstellen.

Dies kann mit Lambdas vermieden werden.

Sie können Parameterpakete wie folgt in eine Reihe von Aufrufen eines Lambda entpacken:

```
template<std::size_t I>
using index_t = std::integral_constant<std::size_t, I>;
template<std::size_t I>
constexpr index_t<I> index{};

template<class=void, std::size_t...Is>
auto index_over( std::index_sequence<Is...> ) {
  return [](auto&& f){
    using discard=int[];
    (void)discard{0,(void(
      f( index<Is> )
    ),0)...};
  };
}
```

```
template<std::size_t N>
auto index_over(index_t<N> = {}) {
  return index_over( std::make_index_sequence<N>{} );
}
```

C ++ 17

Mit fold-Ausdrücken kann `index_over()` folgendermaßen vereinfacht werden:

```
template<class=void, std::size_t...Is>
auto index_over( std::index_sequence<Is...> ) {
  return [](auto&& f){
    ((void)(f(index<Is>)), ...);
  };
}
```

Sobald Sie dies getan haben, können Sie dies verwenden, um das manuelle Auspacken von Parameterpaketen durch eine zweite Überladung in anderem Code zu ersetzen, sodass Sie die Parameterpakete "inline" entpacken können:

```
template<class Tup, class F>
void for_each_tuple_element(Tup&& tup, F&& f) {
  using T = std::remove_reference_t<Tup>;
  using std::tuple_size;
  auto from_zero_to_N = index_over< tuple_size<T>{} >();

  from_zero_to_N(
    [&](auto i){
      using std::get;
      f( get<i>( std::forward<Tup>(tup) ) );
    }
  );
}
```

Das `auto i` durch das `index_over` an das Lambda `index_over` ist `std::integral_constant<std::size_t, ???>` . Dies hat eine `constexpr` Konvertierung in `std::size_t` , die nicht vom Zustand `this` `constexpr` abhängt. `constexpr` können wir es als Kompilierungszeitkonstante verwenden, z. B. wenn wir es an `std::get<i>` .

Um zum Spielzeugbeispiel oben zurückzukehren, schreiben Sie es wie folgt um:

```
template<std::size_t I>
void print_indexes_upto() {
  index_over(index<I>)([](auto i){
    std::cout << i << '\n'; // here i is a compile-time constant
  });
}
```

Das ist viel kürzer und behält die Logik in dem Code, der es verwendet.

Live-Beispiel zum Spielen.

Lambdas online lesen: https://riptutorial.com/de/cplusplus/topic/572/lambdas

Kapitel 71: Layout der Objekttypen

Bemerkungen

Siehe auch Größe der Integraltypen .

Examples

Klassenarten

Mit "Klasse" meinen wir einen Typ, der mit dem Schlüsselwort `class` oder `struct` definiert wurde (nicht jedoch mit `enum struct` oder `enum class`).

- Selbst eine leere Klasse belegt immer noch mindestens ein Byte Speicherplatz. es wird also nur aus Polsterung bestehen. Dies stellt sicher, dass wenn `p` auf ein Objekt einer leeren Klasse zeigt, `p + 1` eine eindeutige Adresse ist und auf ein bestimmtes Objekt zeigt. Es ist jedoch möglich, dass eine leere Klasse eine Größe von 0 hat, wenn sie als Basisklasse verwendet wird. Siehe leere Basisoptimierung .

```
class Empty_1 {};                               // sizeof(Empty_1)       == 1
class Empty_2 {};                               // sizeof(Empty_2)       == 1
class Derived : Empty_1 {};                     // sizeof(Derived)       == 1
class DoubleDerived : Empty_1, Empty_2 {};      // sizeof(DoubleDerived) == 1
class Holder { Empty_1 e; };                    // sizeof(Holder)        == 1
class DoubleHolder { Empty_1 e1; Empty_2 e2; }; // sizeof(DoubleHolder)  == 2
class DerivedHolder : Empty_1 { Empty_1 e; };   // sizeof(DerivedHolder) == 2
```

- Die Objektdarstellung eines Klassentyps enthält die Objektdarstellungen der Basisklassen- und nicht statischen Elementtypen. Daher zum Beispiel in der folgenden Klasse:

```
struct S {
    int x;
    char* y;
};
```

  Es gibt eine aufeinanderfolgende Folge von `sizeof(int)` Bytes innerhalb eines `S` Objekts, das als *Unterobjekt bezeichnet wird und* den Wert von `x` , und ein anderes Subobjekt mit `sizeof(char*)` Bytes, das den Wert von `y` . Die beiden können nicht verschachtelt werden.

- Wenn ein Klassentyp Member und / oder Basisklassen mit den Typen `t1, t2,...tN` , muss die Größe mindestens die `sizeof(t1) + sizeof(t2) + ... + sizeof(tN)` wenn die vorstehenden Punkte angegeben sind . Abhängig von den Ausrichtungsanforderungen der Member und der Basisklassen kann der Compiler jedoch gezwungen sein, die Auffüllung zwischen Unterobjekten oder am Anfang oder Ende des vollständigen Objekts einzufügen.

```
struct AnInt      { int i; };
  // sizeof(AnInt)        == sizeof(int)
```

```
  // Assuming a typical 32- or 64-bit system, sizeof(AnInt)        == 4 (4).
struct TwoInts    { int i, j; };
  // sizeof(TwoInts)      >= 2 * sizeof(int)
  // Assuming a typical 32- or 64-bit system, sizeof(TwoInts)      == 8 (4 + 4).
struct IntAndChar { int i; char c; };
  // sizeof(IntAndChar)   >= sizeof(int) + sizeof(char)
  // Assuming a typical 32- or 64-bit system, sizeof(IntAndChar)   == 8 (4 + 1 +
padding).
struct AnIntDerived : AnInt { long long l; };
  // sizeof(AnIntDerived) >= sizeof(AnInt) + sizeof(long long)
  // Assuming a typical 32- or 64-bit system, sizeof(AnIntDerived) == 16 (4 + padding +
8).
```

- Wenn aufgrund von Ausrichtungsanforderungen ein Abstand in ein Objekt eingefügt wird, ist die Größe größer als die Summe der Größen der Elemente und Basisklassen. Bei einer Ausrichtung von `n` Byte ist size normalerweise das kleinste Vielfache von `n` das größer ist als die Größe aller Member und Basisklassen. Jedes Mitglied `memN` wird typischerweise an einer Adresse angeordnet werden , der ein Vielfaches von ist `alignof(memN)` , und `n` ist typischerweise die größte sein `alignof` out aller Mitglieder `alignof` s. Wenn ein Glied mit einem kleineren `alignof` von einem Glied mit einem größeren `alignof` , besteht daher die Möglichkeit, dass das letztere Glied nicht richtig ausgerichtet wird, wenn es unmittelbar nach dem ersteren platziert wird. In diesem Fall wird eine Polsterung (auch als *Ausrichtungselement bekannt*) zwischen den beiden Elementen angeordnet, so dass das letztere Element seine gewünschte Ausrichtung haben kann. Wenn im `alignof` zu einem `alignof` mit einem größeren `alignof` ein `alignof` mit einem kleineren `alignof` , ist normalerweise kein Auffüllen erforderlich. Dieser Vorgang wird auch als "Packen" bezeichnet.
  Da Klassen normalerweise das `alignof` ihres Mitglieds mit dem größten `alignof` , werden Klassen normalerweise an dem `alignof` des größten eingebauten Typs ausgerichtet, den sie direkt oder indirekt enthalten.

```
// Assume sizeof(short) == 2, sizeof(int) == 4, and sizeof(long long) == 8.
// Assume 4-byte alignment is specified to the compiler.
struct Char { char c; };
  // sizeof(Char)               == 1 (sizeof(char))
struct Int  { int i; };
  // sizeof(Int)                == 4 (sizeof(int))
struct CharInt { char c; int i; };
  // sizeof(CharInt)            == 8 (1 (char) + 3 (padding) + 4 (int))
struct ShortIntCharInt { short s; int i; char c; int j; };
  // sizeof(ShortIntCharInt)    == 16 (2 (short) + 2 (padding) + 4 (int) + 1 (char) +
  //                                   3 (padding) + 4 (int))
struct ShortIntCharCharInt { short s; int i; char c; char d; int j; };
  // sizeof(ShortIntCharCharInt) == 16 (2 (short) + 2 (padding) + 4 (int) + 1 (char) +
  //                                   1 (char) + 2 (padding) + 4 (int))
struct ShortCharShortInt { short s; char c; short t; int i; };
  // sizeof(ShortCharShortInt)  == 12 (2 (short) + 1 (char) + 1 (padding) + 2 (short) +
  //                                   2 (padding) + 4 (int))
struct IntLLInt { int i; long long l; int j; };
  // sizeof(IntLLInt)           == 16 (4 (int) + 8 (long long) + 4 (int))
  // If packing isn't explicitly specified, most compilers will pack this as
  //   8-byte alignment, such that:
  // sizeof(IntLLInt)           == 24 (4 (int) + 4 (padding) + 8 (long long) +
  //                                   4 (int) + 4 (padding))
```

```
// Assume sizeof(bool) == 1, sizeof(ShortIntCharInt) == 16, and sizeof(IntLLInt) == 24.
// Assume default alignment: alignof(ShortIntCharInt) == 4, alignof(IntLLInt) == 8.
struct ShortChar3ArrShortInt {
    short s;
    char c3[3];
    short t;
    int i;
};
  // ShortChar3ArrShortInt has 4-byte alignment: alignof(int) >= alignof(char) &&
  //                                             alignof(int) >= alignof(short)
  // sizeof(ShortChar3ArrShortInt) == 12 (2 (short) + 3 (char[3]) + 1 (padding) +
  //                                      2 (short) + 4 (int))
  // Note that t is placed at alignment of 2, not 4.  alignof(short) == 2.

struct Large_1 {
    ShortIntCharInt sici;
    bool b;
    ShortIntCharInt tjdj;
};
  // Large_1 has 4-byte alignment.
    // alignof(ShortIntCharInt) == alignof(int) == 4
    // alignof(b) == 1
    // Therefore, alignof(Large_1) == 4.
  // sizeof(Large_1) == 36 (16 (ShortIntCharInt) + 1 (bool) + 3 (padding) +
  //                        16 (ShortIntCharInt))
struct Large_2 {
    IntLLInt illi;
    float f;
    IntLLInt jmmj;
};
  // Large_2 has 8-byte alignment.
    // alignof(IntLLInt) == alignof(long long) == 8
    // alignof(float) == 4
    // Therefore, alignof(Large_2) == 8.
  // sizeof(Large_2) == 56 (24 (IntLLInt) + 4 (float) + 4 (padding) + 24 (IntLLInt))
```

C ++ 11

- Wenn strenge Ausrichtung mit `alignas` erzwungen wird, wird der Typ durch Auffüllen mit dem `alignas` gezwungen, die angegebene Ausrichtung zu erfüllen, selbst wenn er sonst kleiner wäre. Bei der folgenden Definition werden in `Chars<5>` am Ende drei (oder möglicherweise mehr) Auffüllbytes eingefügt, so dass die Gesamtgröße 8 beträgt. Es ist nicht möglich, dass eine Klasse mit einer Ausrichtung von 4 eine Größe hat von 5, da es unmöglich ist, ein Array dieser Klasse zu erstellen, so dass die Größe auf ein Vielfaches von 4 "aufgerundet" werden muss, indem Auffüllbytes eingefügt werden.

```
// This type shall always be aligned to a multiple of 4.  Padding shall be inserted as
// needed.
// Chars<1>..Chars<4> are 4 bytes, Chars<5>..Chars<8> are 8 bytes, etc.
template<size_t SZ>
struct alignas(4) Chars { char arr[SZ]; };

static_assert(sizeof(Chars<1>) == sizeof(Chars<4>), "Alignment is strict.\n");
```

- Wenn zwei nicht statische Member einer Klasse denselben Zugriffsbezeichner haben , wird

derjenige, der später in der Deklarationsreihenfolge erscheint, später in der Objektdarstellung garantiert. Wenn jedoch zwei nicht statische Member unterschiedliche Zugriffsspezifizierer haben, ist ihre relative Reihenfolge innerhalb des Objekts nicht spezifiziert.
- Es ist nicht festgelegt, in welcher Reihenfolge die Basisklassen-Unterobjekte in einem Objekt angezeigt werden, ob sie nacheinander auftreten und ob sie vor, nach oder zwischen untergeordneten Unterobjekten erscheinen.

Arithmetische arten

Schmale Zeichentypen

Der `unsigned char` Typ verwendet alle Bits, um eine Binärzahl darzustellen. Wenn also `unsigned char` 8 Bit lang ist, repräsentieren die 256 möglichen Bitmuster eines `char` Objekts die 256 verschiedenen Werte {0, 1, ..., 255}. Die Zahl 42 wird garantiert durch das Bitmuster `00101010` .

Der `signed char` hat keine Auffüllbits, *dh* wenn das `signed char` Zeichen 8 Bit lang ist, hat es eine Kapazität von 8 Bit, um eine Zahl darzustellen.

Beachten Sie, dass diese Garantien nicht für andere Typen als schmale Zeichentypen gelten.

Integer-Typen

Die vorzeichenlosen Integer-Typen verwenden ein reines Binärsystem, können jedoch Füllbits enthalten. Zum Beispiel ist es möglich (wenn auch unwahrscheinlich), dass das `unsigned int` 64 Bit lang ist, aber nur Ganzzahlen zwischen 0 und 2^{32-1} einschließlich speichern kann. Die anderen 32 Bits wären Auffüllbits, die nicht direkt beschrieben werden sollten.

Die vorzeichenbehafteten Integer-Typen verwenden ein binäres System mit Vorzeichenbit und möglicherweise Füllbits. Werte, die zum allgemeinen Bereich eines vorzeichenbehafteten Integer-Typs und des entsprechenden vorzeichenlosen Integer-Typs gehören, haben dieselbe Darstellung. Wenn beispielsweise das Bitmuster `0001010010101011` eines `unsigned short` Objekts `unsigned short` den Wert `5291` , dann repräsentiert es auch den Wert `5291` wenn es als `short` Objekt interpretiert wird.

Es ist implementierungsdefiniert, ob eine Zwei-Komplement-, Ein-Komplement- oder Vorzeichengrößen-Darstellung verwendet wird, da alle drei Systeme die Anforderung des vorherigen Abschnitts erfüllen.

Fließkomma-Typen

Die Wertdarstellung von Gleitkommatypen ist implementierungsdefiniert. Am häufigsten entsprechen die `float` und `double` Typen IEEE 754 und sind 32 und 64 Bit lang (so hätte `float` beispielsweise eine Genauigkeit von 23 Bit, die 8 Exponenten-Bits und 1 Vorzeichen-Bits folgen

würde). Der Standard garantiert jedoch nichts. Gleitkommatypen haben häufig "Trap-Darstellungen", die Fehler verursachen, wenn sie in Berechnungen verwendet werden.

Arrays

Ein Array-Typ hat keine Auffüllung zwischen seinen Elementen. Daher ist ein Array mit dem Elementtyp `T` nur eine Folge von `T` Objekten, die in der Reihenfolge angeordnet sind.

Ein mehrdimensionales Array ist ein Array von Arrays, und das oben Gesagte gilt rekursiv. Zum Beispiel, wenn wir die Erklärung haben

```
int a[5][3];
```

dann `a` ist ein Array von Arrays von 5 3 `int` s. Daher ist `a[0]` , die aus den drei Elementen `a[0][0]` , `a[0][1]` , `a[0][2]` besteht, vor `a[1]` , die besteht, im Speicher abgelegt von `a[1][0]` , `a[1][1]` und `a[1][2]` . Dies wird als *Reihenhauptordnung bezeichnet* .

Layout der Objekttypen online lesen: https://riptutorial.com/de/cplusplus/topic/9329/layout-der-objekttypen

Kapitel 72: Literale

Einführung

Traditionell ist ein Literal ein Ausdruck, der eine Konstante bezeichnet, deren Typ und Wert aus der Schreibweise ersichtlich sind. Zum Beispiel ist `42` ein Literal, während `x` nicht ist, da man seine Deklaration sehen muss, um seinen Typ zu kennen, und vorherige Codezeilen lesen, um seinen Wert zu kennen.

In C ++ 11 wurden jedoch auch benutzerdefinierte Literale hinzugefügt, bei denen es sich nicht um herkömmliche Literale handelt , die jedoch als Abkürzung für Funktionsaufrufe verwendet werden können.

Examples

wahr

Ein Schlüsselwort, das einen der zwei möglichen Werte des Typs `bool` angibt.

```
bool ok = true;
if (!f()) {
    ok = false;
    goto end;
}
```

falsch

Ein Schlüsselwort, das einen der zwei möglichen Werte des Typs `bool` angibt.

```
bool ok = true;
if (!f()) {
    ok = false;
    goto end;
}
```

nullptr

C ++ 11

Ein Schlüsselwort, das eine Nullzeiger-Konstante bezeichnet. Es kann in einen beliebigen Zeiger oder Zeiger-zu-Mitglied-Typ konvertiert werden, der einen Nullzeiger des resultierenden Typs ergibt.

```
Widget* p = new Widget();
delete p;
p = nullptr; // set the pointer to null after deletion
```

Beachten Sie, dass `nullptr` selbst kein Zeiger ist. Der Typ von `nullptr` ist ein grundlegender Typ, der als `std::nullptr_t` .

```
void f(int* p);

template <class T>
void g(T* p);

void h(std::nullptr_t p);

int main() {
    f(nullptr); // ok
    g(nullptr); // error
    h(nullptr); // ok
}
```

diese

Innerhalb einer Member-Funktion einer Klasse ist das Schlüsselwort `this` ein Zeiger auf die Instanz der Klasse, in der die Funktion aufgerufen wurde. `this` kann nicht in einer statischen Memberfunktion verwendet werden.

```
struct S {
    int x;
    S& operator=(const S& other) {
        x = other.x;
        // return a reference to the object being assigned to
        return *this;
    }
};
```

Die Art der `this` hängt von der CV-Qualifikation der Elementfunktion: Wenn `X::f` ist `const` , dann die Art der `this` innerhalb `f` ist `const X*` , so dass `this` nicht verwendet werden , können nicht statischen Datenelementen zu modifizieren , von innerhalb einer `const` Mitgliedsfunktion. Ebenso erbt `this` die `volatile` Qualifizierung von der Funktion, in der es erscheint.

C ++ 11

`this` kann auch in einem *Brace-or-Equalizer* für ein nicht statisches Datenelement verwendet werden.

```
struct S;
struct T {
    T(const S* s);
    // ...
};
struct S {
    // ...
    T t{this};
};
```

`this` ist ein Wert, der nicht zugewiesen werden kann.

Integer-Literal

Ein ganzzahliges Literal ist ein primärer Ausdruck des Formulars

- Dezimal-Literal

Es ist eine Dezimalstelle ungleich Null (1, 2, 3, 4, 5, 6, 7, 8, 9), gefolgt von null oder mehr Dezimalstellen (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

```
int d = 42;
```

- Oktal-Literal

Es ist die Ziffer Null (0) gefolgt von Null oder mehr Oktalstellen (0, 1, 2, 3, 4, 5, 6, 7).

```
int o = 052
```

- Hex-Literal

Es ist die Zeichenfolge 0x oder die Zeichenfolge 0X, gefolgt von einer oder mehreren Hexadezimalziffern (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, A, b, B, c, C) d, D, e, E, f, F)

```
int x = 0x2a; int X = 0X2A;
```

- Binär-Literal (seit C ++ 14)

Es ist die Zeichenfolge 0b oder die Zeichenfolge 0B gefolgt von einer oder mehreren binären Ziffern (0, 1).

```
int b = 0b101010; // C++14
```

Ein Integer-Suffix kann, falls angegeben, eine oder beide der folgenden Angaben enthalten (wenn beide angegeben werden, können sie in beliebiger Reihenfolge angezeigt werden:

- vorzeichenloses Suffix (das Zeichen u oder das Zeichen U)

```
unsigned int u_1 = 42u;
```

- long-Suffix (das Zeichen l oder das Zeichen L) oder das long-long-Suffix (die Zeichenfolge ll oder die Zeichenfolge LL) (seit C ++ 11)

Die folgenden Variablen werden ebenfalls mit demselben Wert initialisiert:

```
unsigned long long l1 = 18446744073709550592ull; // C++11
unsigned long long l2 = 18'446'744'073'709'550'592llu; // C++14
unsigned long long l3 = 1844'6744'0737'0955'0592uLL; // C++14
unsigned long long l4 = 184467'440737'0'95505'92LLU; // C++14
```

Anmerkungen

Buchstaben in den Ganzzahl-Literalen unterscheiden nicht zwischen Groß- und Kleinschreibung: 0xDeAdBaBeU und 0XdeadBABEu stellen die gleiche Zahl dar (eine Ausnahme ist das long-long-Suffix, das entweder ll oder LL ist, niemals ll oder Ll).

Es gibt keine negativen Ganzzahlliterale. Ausdrücke wie -1 wenden den unären Minusoperator auf den durch das Literal dargestellten Wert an, der implizite Typkonvertierungen beinhalten kann.

In C vor C99 (aber nicht in C ++) dürfen nicht gefasste Dezimalwerte, die nicht in long int passen, den Typ unsigned long int haben.

Wenn sie in einem steuernden Ausdruck von #if oder #elif verwendet werden, verhalten sich alle vorzeichenbehafteten Ganzzahlkonstanten so, als hätten sie den Typ std :: intmax_t, und alle vorzeichenlosen Ganzzahlkonstanten verhalten sich so, als hätten sie den Typ std :: uintmax_t.

Literale online lesen: https://riptutorial.com/de/cplusplus/topic/7836/literale

Kapitel 73: Mehrere Werte aus einer Funktion zurückgeben

Einführung

In vielen Situationen ist es nützlich, mehrere Werte aus einer Funktion zurückzugeben: Wenn Sie beispielsweise einen Artikel eingeben und Preis und Anzahl auf Lager zurückgeben möchten, kann diese Funktion hilfreich sein. Es gibt viele Möglichkeiten, dies in C ++ zu tun, und die meisten beinhalten die STL. Wenn Sie die STL jedoch aus irgendeinem Grund vermeiden möchten, gibt es noch verschiedene Möglichkeiten, einschließlich `structs/classes` und `arrays` .

Examples

Ausgabeparameter verwenden

Parameter können verwendet werden, um einen oder mehrere Werte zurückzugeben. Diese Parameter müssen nicht `const` Zeiger oder Verweise sein.

Verweise:

```
void calculate(int a, int b, int& c, int& d, int& e, int& f) {
    c = a + b;
    d = a - b;
    e = a * b;
    f = a / b;
}
```

Zeiger:

```
void calculate(int a, int b, int* c, int* d, int* e, int* f) {
    *c = a + b;
    *d = a - b;
    *e = a * b;
    *f = a / b;
}
```

Einige Bibliotheken oder Frameworks verwenden ein leeres 'OUT' `#define` , um deutlich zu machen, welche Parameter in der Funktionssignatur Ausgabeparameter sind. Dies hat keine funktionalen Auswirkungen und wird kompiliert, macht aber die Funktionssignatur etwas klarer.

```
#define OUT

void calculate(int a, int b, OUT int& c) {
    c = a + b;
}
```

Verwenden von std :: tuple

C ++ 11

Der Typ `std::tuple` kann eine beliebige Anzahl von Werten, die möglicherweise Werte verschiedener Typen enthalten, in einem einzigen Rückgabeobjekt bündeln:

```
std::tuple<int, int, int, int> foo(int a, int b) { // or auto (C++14)
   return std::make_tuple(a + b, a - b, a * b, a / b);
}
```

In C ++ 17 kann eine geschweifte Initialisierungsliste verwendet werden:

C ++ 17

```
std::tuple<int, int, int, int> foo(int a, int b)    {
    return {a + b, a - b, a * b, a / b};
}
```

Das Abrufen von Werten aus dem zurückgegebenen `tuple` kann mühsam sein und erfordert die Verwendung der Vorlagenfunktion `std::get` :

```
auto mrvs = foo(5, 12);
auto add = std::get<0>(mrvs);
auto sub = std::get<1>(mrvs);
auto mul = std::get<2>(mrvs);
auto div = std::get<3>(mrvs);
```

Wenn die Typen deklariert werden können, bevor die Funktion zurückkehrt, kann mit `std::tie` ein `tuple` in vorhandene Variablen entpackt werden:

```
int add, sub, mul, div;
std::tie(add, sub, mul, div) = foo(5, 12);
```

Wenn einer der zurückgegebenen Werte nicht benötigt wird, kann `std::ignore` verwendet werden:

```
std::tie(add, sub, std::ignore, div) = foo(5, 12);
```

C ++ 17

Strukturierte Bindungen können verwendet werden, um `std::tie` zu vermeiden:

```
auto [add, sub, mul, div] = foo(5,12);
```

Wenn Sie ein Tupel von lvalue-Referenzen anstelle eines Tupels von Werten zurückgeben möchten, verwenden Sie `std::tie` anstelle von `std::make_tuple` .

```
std::tuple<int&, int&> minmax( int& a, int& b ) {
  if (b<a)
    return std::tie(b,a);
```

```
  else
    return std::tie(a,b);
}
```

was erlaubt

```
void increase_least(int& a, int& b) {
  std::get<0>(minmax(a,b))++;
}
```

In seltenen Fällen verwenden Sie `std::forward_as_tuple` anstelle von `std::tie` ; Seien Sie vorsichtig, wenn Sie dies tun, da Provisorien möglicherweise nicht lange genug dauern, um aufgebraucht zu werden.

Verwenden von std :: array

C ++ 11

Der Container `std::array` kann eine feste Anzahl von Rückgabewerten zusammenfassen. Diese Nummer muss zur Kompilierzeit bekannt sein und alle Rückgabewerte müssen vom selben Typ sein:

```
std::array<int, 4> bar(int a, int b) {
    return { a + b, a - b, a * b, a / b };
}
```

Dies ersetzt Arrays vom Stil der Form `int bar[4]` . Der Vorteil ist, dass jetzt verschiedene `c++` std-Funktionen verwendet werden können. Es bietet auch nützliche Member - Funktionen wie `at` dem eine sichere Mitglied Zugriffsfunktion mit gebundener Prüfung ist, und eine `size` , die die Größe des Arrays ohne Berechnung zurückkehren kann.

Verwenden von std :: pair

Das struct template `std::pair` kann *genau* zwei Rückgabewerte eines beliebigen Typs zusammenfassen:

```
#include <utility>
std::pair<int, int> foo(int a, int b) {
    return std::make_pair(a+b, a-b);
}
```

Mit C ++ 11 oder höher kann anstelle von `std::make_pair` eine Initialisierungsliste verwendet werden:

C ++ 11

```
#include <utility>
std::pair<int, int> foo(int a, int b) {
    return {a+b, a-b};
}
```

Die einzelnen Werte des zurückgegebenen `std::pair` können mit den `first` und `second` `std::pair` abgerufen werden:

```
std::pair<int, int> mrvs = foo(5, 12);
std::cout << mrvs.first + mrvs.second << std::endl;
```

Ausgabe:

```
10
```

Struct verwenden

Eine `struct` kann verwendet werden, um mehrere Rückgabewerte zu bündeln:

C ++ 11

```
struct foo_return_type {
    int add;
    int sub;
    int mul;
    int div;
};

foo_return_type foo(int a, int b) {
    return {a + b, a - b, a * b, a / b};
}

auto calc = foo(5, 12);
```

C ++ 11

Anstelle der Zuordnung zu einzelnen Feldern kann ein Konstruktor verwendet werden, um das Erstellen von zurückgegebenen Werten zu vereinfachen:

```
struct foo_return_type {
    int add;
    int sub;
    int mul;
    int div;
    foo_return_type(int add, int sub, int mul, int div)
    : add(add), sub(sub), mul(mul), div(div) {}
};

foo_return_type foo(int a, int b) {
     return foo_return_type(a + b, a - b, a * b, a / b);
}

foo_return_type calc = foo(5, 12);
```

Die einzelnen Ergebnisse, die von der Funktion `foo()` können durch Zugriff auf die Member-Variablen des `struct calc` abgerufen werden:

```
std::cout << calc.add << ' ' << calc.sub << ' ' << calc.mul << ' ' << calc.div << '\n';
```

Ausgabe:

17-7 60 0

Hinweis: Wenn Sie eine `struct` , werden die zurückgegebenen Werte in einem einzigen Objekt zusammengefasst und können mit aussagekräftigen Namen aufgerufen werden. Dies hilft auch, die Anzahl der überflüssigen Variablen zu reduzieren, die im Bereich der zurückgegebenen Werte erstellt werden.

C ++ 17

Um eine auszupacken `struct` aus einer Funktion, zurück strukturierte Bindungen können verwendet werden. Dadurch werden die Out-Parameter mit den In-Parametern auf eine Stufe gestellt:

```
int a=5, b=12;
auto[add, sub, mul, div] = foo(a, b);
std::cout << add << ' ' << sub << ' ' << mul << ' ' << div << '\n';
```

Die Ausgabe dieses Codes ist identisch mit der obigen. Die `struct` wird weiterhin verwendet, um die Werte aus der Funktion zurückzugeben. Dadurch können Sie die Felder individuell bearbeiten.

Strukturierte Bindungen

C ++ 17

C ++ 17 führt strukturierte Bindungen ein, die den Umgang mit mehreren Rückgabetypen noch einfacher machen, da Sie sich nicht auf `std::tie()` oder manuelles Tupel-Auspacken verlassen müssen:

```
std::map<std::string, int> m;

// insert an element into the map and check if insertion succeeded
auto [iterator, success] = m.insert({"Hello", 42});

if (success) {
    // your code goes here
}

// iterate over all elements without having to use the cryptic 'first' and 'second' names
for (auto const& [key, value] : m) {
    std::cout << "The value for " << key << " is " << value << '\n';
}
```

Strukturierte Bindungen können standardmäßig mit `std::pair` , `std::tuple` und einem beliebigen Typ verwendet werden, dessen nicht statische Datenmitglieder entweder öffentliche direkte Mitglieder oder Mitglieder einer eindeutigen Basisklasse sind:

```
struct A { int x; };
struct B : A { int y; };
B foo();
```

```
// with structured bindings
const auto [x, y] = foo();

// equivalent code without structured bindings
const auto result = foo();
auto& x = result.x;
auto& y = result.y;
```

Wenn Sie Ihren Typ "tupelartig" machen, arbeitet er automatisch mit Ihrem Typ. Ein tuple-like ist ein Typ mit den entsprechenden `tuple_size` , `tuple_element` und `get tuple_element` :

```
namespace my_ns {
    struct my_type {
        int x;
        double d;
        std::string s;
    };
    struct my_type_view {
        my_type* ptr;
    };
}

namespace std {
    template<>
    struct tuple_size<my_ns::my_type_view> : std::integral_constant<std::size_t, 3>
    {};

    template<> struct tuple_element<my_ns::my_type_view, 0>{ using type = int; };
    template<> struct tuple_element<my_ns::my_type_view, 1>{ using type = double; };
    template<> struct tuple_element<my_ns::my_type_view, 2>{ using type = std::string; };
}

namespace my_ns {
    template<std::size_t I>
    decltype(auto) get(my_type_view const& v) {
        if constexpr (I == 0)
            return v.ptr->x;
        else if constexpr (I == 1)
            return v.ptr->d;
        else if constexpr (I == 2)
            return v.ptr->s;
        static_assert(I < 3, "Only 3 elements");
    }
}
```

das funktioniert jetzt:

```
my_ns::my_type t{1, 3.14, "hello world"};

my_ns::my_type_view foo() {
    return {&t};
}

int main() {
    auto[x, d, s] = foo();
    std::cout << x << ',' << d << ',' << s << '\n';
}
```

Verwenden eines Funktionsobjekt-Consumer

Wir können einen Verbraucher bereitstellen, der mit mehreren relevanten Werten aufgerufen wird:

C ++ 11

```
template <class F>
void foo(int a, int b, F consumer) {
    consumer(a + b, a - b, a * b, a / b);
}

// use is simple... ignoring some results is possible as well
foo(5, 12, [](int sum, int , int , int ){
    std::cout << "sum is " << sum << '\n';
});
```

Dies wird als "Weiterleitungsstil" bezeichnet .

Sie können eine Funktion, die ein Tupel zurückgibt, in eine Continuous Passing-Style-Funktion anpassen:

C ++ 17

```
template<class Tuple>
struct continuation {
  Tuple t;
  template<class F>
  decltype(auto) operator->*(F&& f)&&{
    return std::apply( std::forward<F>(f), std::move(t) );
  }
};
std::tuple<int,int,int,int> foo(int a, int b);

continuation(foo(5,12))->*[](int sum, auto&&...) {
  std::cout << "sum is " << sum << '\n';
};
```

wobei komplexere Versionen in C ++ 14 oder C ++ 11 beschreibbar sind.

Mit std :: vector

Ein `std::vector` kann nützlich sein, um eine dynamische Anzahl von Variablen desselben Typs zurückzugeben. Im folgenden Beispiel wird als Datentyp `int` verwendet, aber ein `std::vector` kann jeden beliebigen Typ enthalten, der trivial kopierbar ist:

```
#include <vector>
#include <iostream>

// the following function returns all integers between and including 'a' and 'b' in a vector
// (the function can return up to std::vector::max_size elements with the vector, given that
// the system's main memory can hold that many items)
std::vector<int> fillVectorFrom(int a, int b) {
    std::vector<int> temp;
    for (int i = a; i <= b; i++) {
```

```
        temp.push_back(i);
    }
    return temp;
}

int main() {
    // assigns the filled vector created inside the function to the new vector 'v'
    std::vector<int> v = fillVectorFrom(1, 10);

    // prints "1 2 3 4 5 6 7 8 9 10 "
    for (int i = 0; i < v.size(); i++) {
        std::cout << v[i] << " ";
    }
    std::cout << std::endl;
    return 0;
}
```

Ausgabe-Iterator verwenden

Mehrere Werte desselben Typs können zurückgegeben werden, indem ein Ausgabe-Iterator an die Funktion übergeben wird. Dies ist besonders häufig bei generischen Funktionen (wie bei den Algorithmen der Standardbibliothek).

Beispiel:

```
template<typename Incrementable, typename OutputIterator>
void generate_sequence(Incrementable from, Incrementable to, OutputIterator output) {
    for (Incrementable k = from; k != to; ++k)
        *output++ = k;
}
```

Verwendungsbeispiel:

```
std::vector<int> digits;
generate_sequence(0, 10, std::back_inserter(digits));
// digits now contains {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

Mehrere Werte aus einer Funktion zurückgeben online lesen:
https://riptutorial.com/de/cplusplus/topic/487/mehrere-werte-aus-einer-funktion-zuruckgeben

Kapitel 74: Metaprogrammierung

Einführung

In C ++ bezieht sich Metaprogrammierung auf die Verwendung von Makros oder Vorlagen zum Generieren von Code zur Kompilierzeit.

Im Allgemeinen werden Makros in dieser Rolle nicht gern gesehen, und Vorlagen werden bevorzugt, obwohl sie nicht so generisch sind.

Bei der Template-Metaprogrammierung werden häufig Berechnungen zur Kompilierzeit verwendet, `constexpr` über Templates oder über `constexpr` Funktionen, um die `constexpr` zu erreichen. `constexpr` -Berechnungen sind jedoch keine Metaprogrammierung per se.

Bemerkungen

Metaprogrammierung (oder genauer: Metaprogrammierung von Vorlagen) ist die Praxis, Vorlagen zum Erstellen von Konstanten, Funktionen oder Datenstrukturen zur Kompilierzeit zu verwenden. Dies ermöglicht, dass Berechnungen einmal zur Kompilierungszeit und nicht zu jeder Laufzeit ausgeführt werden.

Examples

Berechnungsfaktoren

Factorials können zur Kompilierzeit unter Verwendung von Schablonenmetaprogrammiertechniken berechnet werden.

```
#include <iostream>

template<unsigned int n>
struct factorial
{
    enum
    {
        value = n * factorial<n - 1>::value
    };
};

template<>
struct factorial<0>
{
    enum { value = 1 };
};

int main()
{
    std::cout << factorial<7>::value << std::endl;    // prints "5040"
}
```

`factorial` ist eine Struktur, aber in der Metaprogrammierung von Vorlagen wird sie als Meta-Funktion der Vorlage behandelt. Konventionell werden Vorlagen-Metafunktionen ausgewertet, indem ein bestimmtes Member überprüft wird, entweder `::type` auf Metafunktionen, die zu Typen führen, oder `::value` für Metafunktionen, die Werte generieren.

Im obigen Code evaluieren wir die `factorial` Metafunktion, indem wir die Vorlage mit den Parametern, die wir übergeben möchten, instantiieren und `::value` , um das Ergebnis der Bewertung zu erhalten.

Die Metafunktion selbst ist darauf angewiesen, dieselbe Metafunktion rekursiv mit kleineren Werten zu instanziieren. Die `factorial<0>` Spezialisierung `factorial<0>` repräsentiert die Beendigungsbedingung. Die Metaprogrammierung von Vorlagen hat die meisten Einschränkungen einer funktionalen Programmiersprache. Rekursion ist also das primäre Konstrukt "Schleife".

Da Vorlagen-Metafunktionen zur Kompilierzeit ausgeführt werden, können ihre Ergebnisse in Kontexten verwendet werden, die Werte für die Kompilierungszeit erfordern. Zum Beispiel:

```
int my_array[factorial<5>::value];
```

Automatische Arrays müssen eine für die Kompilierzeit definierte Größe haben. Das Ergebnis einer Metafunktion ist eine Konstante zur Kompilierzeit, die hier verwendet werden kann.

Einschränkung : Die meisten Compiler erlauben keine Rekursionstiefe über ein Limit hinaus. Beispielsweise hängt die Rekursion von `g++` Compiler standardmäßig von 256 Stufen ab. Im Fall von `g++` kann der Programmierer die Rekursionstiefe mit der Option `-ftemplate-depth-X` .

C ++ 11

Seit C ++ 11 kann die Vorlage `std::integral_constant` für diese Art von Vorlagenberechnung verwendet werden:

```
#include <iostream>
#include <type_traits>

template<long long n>
struct factorial :
  std::integral_constant<long long, n * factorial<n - 1>::value> {};

template<>
struct factorial<0> :
  std::integral_constant<long long, 1> {};

int main()
{
    std::cout << factorial<7>::value << std::endl;    // prints "5040"
}
```

Darüber `constexpr` Funktionen von `constexpr` zu einer saubereren Alternative.

```
#include <iostream>
```

```
constexpr long long factorial(long long n)
{
  return (n == 0) ? 1 : n * factorial(n - 1);
}

int main()
{
  char test[factorial(3)];
  std::cout << factorial(7) << '\n';
}
```

Der Rumpf von `factorial()` wird als einzelne Anweisung geschrieben, da `constexpr` Funktionen in C ++ 11 nur eine recht begrenzte Teilmenge der Sprache verwenden können.

C ++ 14

Seit C ++ 14 wurden viele Einschränkungen für `constexpr` Funktionen `constexpr` und sie können jetzt viel bequemer geschrieben werden:

```
constexpr long long factorial(long long n)
{
  if (n == 0)
    return 1;
  else
    return n * factorial(n - 1);
}
```

Oder auch:

```
constexpr long long factorial(int n)
{
  long long result = 1;
  for (int i = 1; i <= n; ++i) {
    result *= i;
  }
  return result;
}
```

C ++ 17

Seit C ++ 17 kann man den fold-Ausdruck verwenden, um die Fakultät zu berechnen:

```
#include <iostream>
#include <utility>

template <class T, T N, class I = std::make_integer_sequence<T, N>>
struct factorial;

template <class T, T N, T... Is>
struct factorial<T,N,std::index_sequence<T, Is...>> {
   static constexpr T value = (static_cast<T>(1) * ... * (Is + 1));
};

int main() {
   std::cout << factorial<int, 5>::value << std::endl;
```

```
}
```

Iteration über ein Parameterpaket

Häufig müssen wir für jedes Element in einem variadischen Vorlagenparameterpaket eine Operation ausführen. Es gibt viele Möglichkeiten, dies zu tun, und die Lösungen lassen sich mit C ++ 17 leichter lesen und schreiben. Angenommen, wir möchten einfach jedes Element in einer Packung drucken. Die einfachste Lösung ist die Wiederholung:

C ++ 11

```
void print_all(std::ostream& os) {
    // base case
}

template <class T, class... Ts>
void print_all(std::ostream& os, T const& first, Ts const&... rest) {
    os << first;

    print_all(os, rest...);
}
```

Wir können stattdessen den Expander-Trick verwenden, um das Streaming in einer einzigen Funktion auszuführen. Dies hat den Vorteil, dass keine zweite Überlastung erforderlich ist, hat jedoch den Nachteil, dass die Lesbarkeit weniger als der Stern ist:

C ++ 11

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    using expander = int[];
    (void)expander{0,
        (void(os << args), 0)...
    };
}
```

Eine Erklärung, wie das funktioniert, finden Sie in der ausgezeichneten Antwort von TC .

C ++ 17

Mit C ++ 17 verfügen wir über zwei leistungsstarke neue Tools, um dieses Problem zu lösen. Der erste ist ein Falten-Ausdruck:

```
template <class... Ts>
void print_all(std::ostream& os, Ts const&... args) {
    ((os << args), ...);
}
```

Und das zweite ist `if constexpr` , mit dem wir unsere ursprüngliche rekursive Lösung in einer einzigen Funktion schreiben können:

```
template <class T, class... Ts>
```

```
void print_all(std::ostream& os, T const& first, Ts const&... rest) {
    os << first;

    if constexpr (sizeof...(rest) > 0) {
        // this line will only be instantiated if there are further
        // arguments. if rest... is empty, there will be no call to
        // print_all(os).
        print_all(os, rest...);
    }
}
```

Iteration mit std :: integer_sequence

Seit C ++ 14 stellt der Standard die Klassenvorlage bereit

```
template <class T, T... Ints>
class integer_sequence;

template <std::size_t... Ints>
using index_sequence = std::integer_sequence<std::size_t, Ints...>;
```

und eine generierende Metafunktion dafür:

```
template <class T, T N>
using make_integer_sequence = std::integer_sequence<T, /* a sequence 0, 1, 2, ..., N-1 */ >;

template<std::size_t N>
using make_index_sequence = make_integer_sequence<std::size_t, N>;
```

Während dies in C ++ 14 Standard ist, kann dies mit C ++ 11-Tools implementiert werden.

Wir können dieses Tool verwenden, um eine Funktion mit einem `std::tuple` von Argumenten aufzurufen (standardisiert in C ++ 17 als `std::apply`):

```
namespace detail {
    template <class F, class Tuple, std::size_t... Is>
    decltype(auto) apply_impl(F&& f, Tuple&& tpl, std::index_sequence<Is...> ) {
        return std::forward<F>(f)(std::get<Is>(std::forward<Tuple>(tpl))...);
    }
}

template <class F, class Tuple>
decltype(auto) apply(F&& f, Tuple&& tpl) {
    return detail::apply_impl(std::forward<F>(f),
        std::forward<Tuple>(tpl),
        std::make_index_sequence<std::tuple_size<std::decay_t<Tuple>>::value>{});
}


// this will print 3
int f(int, char, double);

auto some_args = std::make_tuple(42, 'x', 3.14);
int r = apply(f, some_args); // calls f(42, 'x', 3.14)
```

Tag-Versand

Eine einfache Möglichkeit, zwischen Funktionen zur Kompilierzeit zu wählen, besteht darin, eine Funktion an ein überladenes Funktionspaar zu übergeben, das ein Tag als ein (normalerweise das letzte) Argument nimmt. Um beispielsweise `std::advance()` zu implementieren, können wir die Iteratorkategorie abschicken:

```
namespace details {
    template <class RAIter, class Distance>
    void advance(RAIter& it, Distance n, std::random_access_iterator_tag) {
        it += n;
    }

    template <class BidirIter, class Distance>
    void advance(BidirIter& it, Distance n, std::bidirectional_iterator_tag) {
        if (n > 0) {
            while (n--) ++it;
        }
        else {
            while (n++) --it;
        }
    }

    template <class InputIter, class Distance>
    void advance(InputIter& it, Distance n, std::input_iterator_tag) {
        while (n--) {
            ++it;
        }
    }
}

template <class Iter, class Distance>
void advance(Iter& it, Distance n) {
    details::advance(it, n,
            typename std::iterator_traits<Iter>::iterator_category{} );
}
```

Die Argumente `std::XY_iterator_tag` der überladenen `details::advance` Funktionen sind nicht verwendete Funktionsparameter. Die tatsächliche Implementierung spielt keine Rolle (eigentlich ist sie völlig leer). Ihr einziger Zweck besteht darin, dem Compiler zu ermöglichen, eine Überladung auszuwählen, die darauf basiert, mit welcher Tag-Klasse `details::advance` aufgerufen wird.

In diesem Beispiel `advance` verwendet die `iterator_traits<T>::iterator_category` metafunction , die eine der zurück `iterator_tag` Klassen, auf dem tatsächlichen Typ abhängig `Iter` . Ein standardmäßig erstelltes Objekt des `iterator_category<Iter>::type` lässt den Compiler dann eine der verschiedenen Überladungen von `details::advance` auswählen. (Dieser Funktionsparameter wird wahrscheinlich vollständig wegoptimiert, da er ein standardmäßig erstelltes Objekt einer leeren `struct` und niemals verwendet wird.)

Durch das Tag-Dispatching erhalten Sie Code, der viel einfacher zu lesen ist als die Äquivalente, die SFINAE und `enable_if` .

Anmerkung: `if constexpr` C ++ 17 die Implementierung von `advance` insbesondere durch `if`

`constexpr` vereinfacht, ist es im Gegensatz zum Tag-Dispatching nicht für offene Implementierungen geeignet.

Ermitteln Sie, ob der Ausdruck gültig ist

Es kann ermittelt werden, ob ein Operator oder eine Funktion für einen Typ aufgerufen werden kann. Um zu testen, ob eine Klasse eine Überladung von `std::hash` , können Sie `std::hash` tun:

```
#include <functional> // for std::hash
#include <type_traits> // for std::false_type and std::true_type
#include <utility> // for std::declval

template<class, class = void>
struct has_hash
    : std::false_type
{};

template<class T>
struct has_hash<T, decltype(std::hash<T>()(std::declval<T>()), void())>
    : std::true_type
{};
```

C ++ 17

Seit C ++ 17 kann `std::void_t` verwendet werden, um diesen Konstrukttyp zu vereinfachen

```
#include <functional> // for std::hash
#include <type_traits> // for std::false_type, std::true_type, std::void_t
#include <utility> // for std::declval

template<class, class = std::void_t<> >
struct has_hash
    : std::false_type
{};

template<class T>
struct has_hash<T, std::void_t< decltype(std::hash<T>()(std::declval<T>())) > >
    : std::true_type
{};
```

Dabei ist `std::void_t` definiert als:

```
template< class... > using void_t = void;
```

Um festzustellen, ob ein Operator wie `operator<` definiert ist, ist die Syntax fast gleich:

```
template<class, class = void>
struct has_less_than
    : std::false_type
{};

template<class T>
struct has_less_than<T, decltype(std::declval<T>() < std::declval<T>(), void())>
    : std::true_type
{};
```

Diese können verwendet werden, um eine `std::unordered_map<T>` wenn `T` eine Überladung für `std::hash` , aber ansonsten versuchen Sie eine `std::map<T>` :

```
template <class K, class V>
using hash_invariant_map = std::conditional_t<
    has_hash<K>::value,
    std::unordered_map<K, V>,
    std::map<K,V>>;
```

Rechenleistung mit C ++ 11 (und höher)

Mit C ++ 11 und höher können Berechnungen zur Kompilierzeit wesentlich einfacher sein. Zum Beispiel wird die Leistung einer bestimmten Anzahl zur Kompilierzeit berechnet:

```
template <typename T>
constexpr T calculatePower(T value, unsigned power) {
    return power == 0 ? 1 : value * calculatePower(value, power-1);
}
```

Das Schlüsselwort `constexpr` ist für die Berechnung der Funktion in der Kompilierungszeit verantwortlich. Dann und nur dann, wenn alle Voraussetzungen dafür erfüllt sind (siehe mehr unter constexpr-Schlüsselwortreferenz), müssen beispielsweise alle Argumente zur Kompilierzeit bekannt sein.

Hinweis: In C ++ 11 darf die `constexpr` Funktion nur aus einer return-Anweisung bestehen.

Vorteile: Vergleicht man dies mit der Standardmethode der Kompilierzeitberechnung, ist diese Methode auch für Laufzeitberechnungen hilfreich. Das bedeutet, dass, wenn die Argumente der Funktion zum Zeitpunkt der Kompilierung nicht bekannt sind (z. B. Wert und Leistung als Eingabe über den Benutzer angegeben werden), die Funktion in einer Kompilierungszeit ausgeführt wird, so dass kein Code kopiert werden muss (wie wir würde in älteren Standards von C ++ erzwungen werden).

Z.B

```
void useExample() {
    constexpr int compileTimeCalculated = calculatePower(3, 3); // computes at compile time,
                                // as both arguments are known at compilation time
                                // and used for a constant expression.
    int value;
    std::cin >> value;
    int runtimeCalculated = calculatePower(value, 3);  // runtime calculated,
                                     // because value is known only at runtime.
}
```

C ++ 17

Eine andere Methode zum Berechnen der Leistung zum Kompilierungszeitpunkt kann den fold-Ausdruck folgendermaßen verwenden:

```
#include <iostream>
```

```
#include <utility>

template <class T, T V, T N, class I = std::make_integer_sequence<T, N>>
struct power;

template <class T, T V, T N, T... Is>
struct power<T, V, N, std::integer_sequence<T, Is...>> {
   static constexpr T value = (static_cast<T>(1) * ... * (V * static_cast<bool>(Is + 1)));
};

int main() {
   std::cout << power<int, 4, 2>::value << std::endl;
}
```

Manuelle Unterscheidung der Typen bei gegebenem Typ T

Bei der Implementierung von SFINAE mithilfe von `std::enable_if` ist es häufig hilfreich, auf `std::enable_if` zuzugreifen, die bestimmen, ob ein bestimmter Typ `T` mit einem Satz von Kriterien übereinstimmt.

Um uns dabei zu helfen, sieht der Standard bereits zwei Typen Analog `true` und `false` , die sind `std::true_type` und `std::false_type` .

Das folgende Beispiel zeigt , wie zu erkennen , ob ein Typ `T` ein Zeiger ist oder nicht, die `is_pointer` Vorlage das Verhalten der Standard imitieren `std::is_pointer` Helfer:

```
template <typename T>
struct is_pointer_: std::false_type {};

template <typename T>
struct is_pointer_<T*>: std::true_type {};

template <typename T>
struct is_pointer: is_pointer_<typename std::remove_cv<T>::type> { }
```

Der obige Code besteht aus drei Schritten (manchmal benötigen Sie nur zwei):

1. Die erste Deklaration von `is_pointer_` ist der *Standardfall* und erbt von `std::false_type` . Der *Standardfall* sollte immer von `std::false_type` erben, da er einer " `false` Bedingung" entspricht.

2. In der zweiten Deklaration wird das `is_pointer_` Template für den Zeiger `T*` ohne sich darum zu kümmern, was `T` wirklich ist. Diese Version erbt von `std::true_type` .

3. Die dritte Deklaration (die echte) entfernt einfach alle unnötigen Informationen aus `T` (in diesem Fall entfernen wir `const` und `volatile` Qualifiers) und greifen dann auf eine der beiden vorherigen Deklarationen zurück.

Da `is_pointer<T>` eine Klasse ist, müssen Sie zum Zugriff auf den Wert entweder:

- Verwenden Sie `::value` , z. B. `is_pointer<int>::value` - `value` ist ein statischer Klassenmitglied vom Typ `bool` das von `std::true_type` oder `std::false_type` .

- Konstruieren Sie ein Objekt dieses Typs, z. B. `is_pointer<int>{}` - Dies funktioniert, weil `std::is_pointer` seinen Standardkonstruktor von `std::true_type` oder `std::false_type` (die über `constexpr` Konstruktoren verfügen) sowie `std::true_type` und `std::false_type` `std::true_type` `std::false_type` hat `constexpr` Konvertierungsoperatoren in `bool` .

Es ist eine gute Angewohnheit, "Helfer-Helfer-Vorlagen" bereitzustellen, mit denen Sie direkt auf den Wert zugreifen können:

```
template <typename T>
constexpr bool is_pointer_v = is_pointer<T>::value;
```

C ++ 17

In C ++ 17 und höher bieten die meisten Helper-Templates bereits eine `_v` Version, z.

```
template< class T > constexpr bool is_pointer_v = is_pointer<T>::value;
template< class T > constexpr bool is_reference_v = is_reference<T>::value;
```

Wenn-dann-sonst

C ++ 11

Der Typ `std::conditional` im Header der Standardbibliothek `<type_traits>` kann den einen oder den anderen Typ auswählen, basierend auf einem booleschen Wert der Kompilierungszeit:

```
template<typename T>
struct ValueOrPointer
{
    typename std::conditional<(sizeof(T) > sizeof(void*)), T*, T>::type vop;
};
```

Diese Struktur enthält einen Zeiger auf `T` wenn `T` größer als die Größe eines Zeigers ist, oder `T` selbst, wenn er kleiner oder gleich der Größe eines Zeigers ist. Deshalb ist `sizeof(ValueOrPointer)` immer <= `sizeof(void*)` .

Generisches Min / Max mit variabler Argumentanzahl

C ++ 11

Es ist möglich, eine generische Funktion (z. B. `min`) zu schreiben, die verschiedene numerische Typen und die Anzahl beliebiger Argumente durch Template-Meta-Programmierung akzeptiert. Diese Funktion deklariert ein `min` für zwei Argumente und rekursiv für mehr.

```
template <typename T1, typename T2>
auto min(const T1 &a, const T2 &b)
-> typename std::common_type<const T1&, const T2&>::type
{
    return a < b ? a : b;
}

template <typename T1, typename T2, typename ... Args>
```

```
auto min(const T1 &a, const T2 &b, const Args& ... args)
-> typename std::common_type<const T1&, const T2&, const Args& ...>::type
{
    return min(min(a, b), args...);
}

auto minimum = min(4, 5.8f, 3, 1.8, 3, 1.1, 9);
```

Metaprogrammierung online lesen:
https://riptutorial.com/de/cplusplus/topic/462/metaprogrammierung

Kapitel 75: Mutexe

Bemerkungen

Es ist besser, *std :: shared_mutex* als *std :: shared_timed_mutex zu verwenden* .

Der Leistungsunterschied ist mehr als doppelt so hoch.

Wenn Sie RWLock verwenden möchten, gibt es zwei Optionen.
Es ist std :: shared_mutex und shared_timed_mutex.
Sie denken vielleicht, dass std :: shared_timed_mutex nur die Version 'std :: shared_mutex + time method' ist.

Die Implementierung ist jedoch völlig anders.

Der folgende Code ist die MSVC14.1-Implementierung von std :: shared_mutex.

```
class shared_mutex
{
public:
typedef _Smtx_t * native_handle_type;

shared_mutex() _NOEXCEPT
    : _Myhandle(0)
    {    // default construct
    }

~shared_mutex() _NOEXCEPT
    {    // destroy the object
    }

void lock() _NOEXCEPT
    {    // lock exclusive
    _Smtx_lock_exclusive(&_Myhandle);
    }

bool try_lock() _NOEXCEPT
    {    // try to lock exclusive
    return (_Smtx_try_lock_exclusive(&_Myhandle) != 0);
    }

void unlock() _NOEXCEPT
    {    // unlock exclusive
    _Smtx_unlock_exclusive(&_Myhandle);
    }

void lock_shared() _NOEXCEPT
```

```
    {    // lock non-exclusive
    _Smtx_lock_shared(&_Myhandle);
    }

bool try_lock_shared() _NOEXCEPT
    {    // try to lock non-exclusive
    return (_Smtx_try_lock_shared(&_Myhandle) != 0);
    }

void unlock_shared() _NOEXCEPT
    {    // unlock non-exclusive
    _Smtx_unlock_shared(&_Myhandle);
    }

native_handle_type native_handle() _NOEXCEPT
    {    // get native handle
    return (&_Myhandle);
    }

shared_mutex(const shared_mutex&) = delete;
shared_mutex& operator=(const shared_mutex&) = delete;
private:
    _Smtx_t _Myhandle;
};

void __cdecl _Smtx_lock_exclusive(_Smtx_t * smtx)
{    /* lock shared mutex exclusively */
AcquireSRWLockExclusive(reinterpret_cast<PSRWLOCK>(smtx));
}

void __cdecl _Smtx_lock_shared(_Smtx_t * smtx)
{    /* lock shared mutex non-exclusively */
AcquireSRWLockShared(reinterpret_cast<PSRWLOCK>(smtx));
}

int __cdecl _Smtx_try_lock_exclusive(_Smtx_t * smtx)
{    /* try to lock shared mutex exclusively */
return (TryAcquireSRWLockExclusive(reinterpret_cast<PSRWLOCK>(smtx)));
}

int __cdecl _Smtx_try_lock_shared(_Smtx_t * smtx)
{    /* try to lock shared mutex non-exclusively */
return (TryAcquireSRWLockShared(reinterpret_cast<PSRWLOCK>(smtx)));
}

void __cdecl _Smtx_unlock_exclusive(_Smtx_t * smtx)
{    /* unlock exclusive shared mutex */
ReleaseSRWLockExclusive(reinterpret_cast<PSRWLOCK>(smtx));
}

void __cdecl _Smtx_unlock_shared(_Smtx_t * smtx)
{    /* unlock non-exclusive shared mutex */
ReleaseSRWLockShared(reinterpret_cast<PSRWLOCK>(smtx));
}
```

Sie sehen, dass std :: shared_mutex in Windows Slim Reader / Write Locks implementiert ist (https://msdn.microsoft.com/ko-kr/library/windows/desktop/aa904937(v=vs.85).aspx)

Betrachten wir nun die Implementierung von std :: shared_timed_mutex.

Der folgende Code ist die MSVC14.1-Implementierung von std :: shared_timed_mutex.

```
class shared_timed_mutex
{
typedef unsigned int _Read_cnt_t;
static constexpr _Read_cnt_t _Max_readers = _Read_cnt_t(-1);
public:
shared_timed_mutex() _NOEXCEPT
    : _Mymtx(), _Read_queue(), _Write_queue(),
        _Readers(0), _Writing(false)
    {    // default construct
    }

~shared_timed_mutex() _NOEXCEPT
    {    // destroy the object
    }

void lock()
    {    // lock exclusive
    unique_lock<mutex> _Lock(_Mymtx);
    while (_Writing)
        _Write_queue.wait(_Lock);
    _Writing = true;
    while (0 < _Readers)
        _Read_queue.wait(_Lock);    // wait for writing, no readers
    }

bool try_lock()
    {    // try to lock exclusive
    lock_guard<mutex> _Lock(_Mymtx);
    if (_Writing || 0 < _Readers)
        return (false);
    else
        {    // set writing, no readers
        _Writing = true;
        return (true);
        }
    }

template<class _Rep,
    class _Period>
    bool try_lock_for(
        const chrono::duration<_Rep, _Period>& _Rel_time)
    {    // try to lock for duration
    return (try_lock_until(chrono::steady_clock::now() + _Rel_time));
    }

template<class _Clock,
    class _Duration>
    bool try_lock_until(
        const chrono::time_point<_Clock, _Duration>& _Abs_time)
    {    // try to lock until time point
    auto _Not_writing = [this] { return (!_Writing); };
    auto _Zero_readers = [this] { return (_Readers == 0); };
    unique_lock<mutex> _Lock(_Mymtx);

    if (!_Write_queue.wait_until(_Lock, _Abs_time, _Not_writing))
        return (false);
```

```
    _Writing = true;

    if (!_Read_queue.wait_until(_Lock, _Abs_time, _Zero_readers))
        {    // timeout, leave writing state
        _Writing = false;
        _Lock.unlock();    // unlock before notifying, for efficiency
        _Write_queue.notify_all();
        return (false);
        }

    return (true);
    }

void unlock()
    {    // unlock exclusive
        {    // unlock before notifying, for efficiency
        lock_guard<mutex> _Lock(_Mymtx);

        _Writing = false;
        }

    _Write_queue.notify_all();
    }

void lock_shared()
    {    // lock non-exclusive
    unique_lock<mutex> _Lock(_Mymtx);
    while (_Writing || _Readers == _Max_readers)
        _Write_queue.wait(_Lock);
    ++_Readers;
    }

bool try_lock_shared()
    {    // try to lock non-exclusive
    lock_guard<mutex> _Lock(_Mymtx);
    if (_Writing || _Readers == _Max_readers)
        return (false);
    else
        {    // count another reader
        ++_Readers;
        return (true);
        }
    }

template<class _Rep,
    class _Period>
    bool try_lock_shared_for(
        const chrono::duration<_Rep, _Period>& _Rel_time)
    {    // try to lock non-exclusive for relative time
    return (try_lock_shared_until(_Rel_time
        + chrono::steady_clock::now()));
    }

template<class _Time>
    bool _Try_lock_shared_until(_Time _Abs_time)
    {    // try to lock non-exclusive until absolute time
    auto _Can_acquire = [this] {
        return (!_Writing && _Readers < _Max_readers); };

    unique_lock<mutex> _Lock(_Mymtx);
```

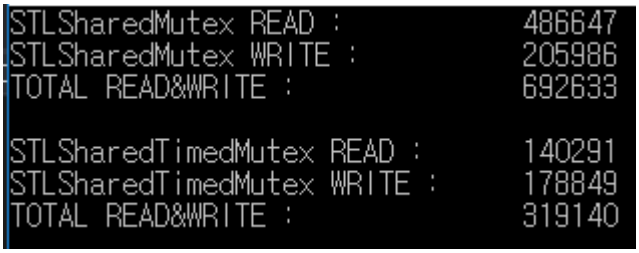
```
    if (!_Write_queue.wait_until(_Lock, _Abs_time, _Can_acquire))
        return (false);

    ++_Readers;
    return (true);
    }

template<class _Clock,
    class _Duration>
    bool try_lock_shared_until(
        const chrono::time_point<_Clock, _Duration>& _Abs_time)
    {    // try to lock non-exclusive until absolute time
    return (_Try_lock_shared_until(_Abs_time));
    }

bool try_lock_shared_until(const xtime *_Abs_time)
    {    // try to lock non-exclusive until absolute time
    return (_Try_lock_shared_until(_Abs_time));
    }

void unlock_shared()
    {    // unlock non-exclusive
    _Read_cnt_t _Local_readers;
    bool _Local_writing;

        {    // unlock before notifying, for efficiency
        lock_guard<mutex> _Lock(_Mymtx);
        --_Readers;
        _Local_readers = _Readers;
        _Local_writing = _Writing;
        }

    if (_Local_writing && _Local_readers == 0)
        _Read_queue.notify_one();
    else if (!_Local_writing && _Local_readers == _Max_readers - 1)
        _Write_queue.notify_all();
    }

shared_timed_mutex(const shared_timed_mutex&) = delete;
shared_timed_mutex& operator=(const shared_timed_mutex&) = delete;
private:
mutex _Mymtx;
condition_variable _Read_queue, _Write_queue;
_Read_cnt_t _Readers;
bool _Writing;
};

class stl_condition_variable_win7 final : public stl_condition_variable_interface
{
public:
    stl_condition_variable_win7()
    {
        __crtInitializeConditionVariable(&m_condition_variable);
    }

    ~stl_condition_variable_win7() = delete;
    stl_condition_variable_win7(const stl_condition_variable_win7&) = delete;
    stl_condition_variable_win7& operator=(const stl_condition_variable_win7&) = delete;

    virtual void destroy() override {}
```

```
    virtual void wait(stl_critical_section_interface *lock) override
    {
        if (!stl_condition_variable_win7::wait_for(lock, INFINITE))
            std::terminate();
    }

    virtual bool wait_for(stl_critical_section_interface *lock, unsigned int timeout) override
    {
        return __crtSleepConditionVariableSRW(&m_condition_variable,
static_cast<stl_critical_section_win7 *>(lock)->native_handle(), timeout, 0) != 0;
    }

    virtual void notify_one() override
    {
        __crtWakeConditionVariable(&m_condition_variable);
    }

    virtual void notify_all() override
    {
        __crtWakeAllConditionVariable(&m_condition_variable);
    }

private:
    CONDITION_VARIABLE m_condition_variable;
};
```

Sie können sehen, dass std :: shared_timed_mutex in std :: condition_value implementiert ist.

Das ist ein großer Unterschied.

Lassen Sie uns die Leistung von zwei von ihnen überprüfen.



Dies ist das Ergebnis eines Lese- / Schreibtests für 1000 Millisekunden.

std :: shared_mutex verarbeitete Lesen / Schreiben um mehr als das Doppelte als std :: shared_timed_mutex.

In diesem Beispiel ist das Lese- / Schreibverhältnis das gleiche, aber die Leserate ist häufiger als die reale Schreibrate.
Daher kann der Leistungsunterschied größer sein.

Der Code unten ist der Code in diesem Beispiel.

```
void useSTLSharedMutex()
{
    std::shared_mutex shared_mtx_lock;
```

```
    std::vector<std::thread> readThreads;
    std::vector<std::thread> writeThreads;

    std::list<int> data = { 0 };
    volatile bool exit = false;

    std::atomic<int> readProcessedCnt(0);
    std::atomic<int> writeProcessedCnt(0);

    for (unsigned int i = 0; i < std::thread::hardware_concurrency(); i++)
    {

        readThreads.push_back(std::thread([&data, &exit, &shared_mtx_lock,
&readProcessedCnt]() {
            std::list<int> mydata;
            int localProcessCnt = 0;

            while (true)
            {
                shared_mtx_lock.lock_shared();

                mydata.push_back(data.back());
                ++localProcessCnt;

                shared_mtx_lock.unlock_shared();

                if (exit)
                    break;
            }

            std::atomic_fetch_add(&readProcessedCnt, localProcessCnt);

        }));

        writeThreads.push_back(std::thread([&data, &exit, &shared_mtx_lock,
&writeProcessedCnt]() {

            int localProcessCnt = 0;

            while (true)
            {
                shared_mtx_lock.lock();

                data.push_back(rand() % 100);
                ++localProcessCnt;

                shared_mtx_lock.unlock();

                if (exit)
                    break;
            }

            std::atomic_fetch_add(&writeProcessedCnt, localProcessCnt);

        }));
    }

    std::this_thread::sleep_for(std::chrono::milliseconds(MAIN_WAIT_MILLISECONDS));
    exit = true;
```

```
    for (auto &r : readThreads)
        r.join();

    for (auto &w : writeThreads)
        w.join();

    std::cout << "STLSharedMutex READ :           " << readProcessedCnt << std::endl;
    std::cout << "STLSharedMutex WRITE :          " << writeProcessedCnt << std::endl;
    std::cout << "TOTAL READ&WRITE :              " << readProcessedCnt + writeProcessedCnt <<
std::endl << std::endl;
}

void useSTLSharedTimedMutex()
{
    std::shared_timed_mutex shared_mtx_lock;

    std::vector<std::thread> readThreads;
    std::vector<std::thread> writeThreads;

    std::list<int> data = { 0 };
    volatile bool exit = false;

    std::atomic<int> readProcessedCnt(0);
    std::atomic<int> writeProcessedCnt(0);

    for (unsigned int i = 0; i < std::thread::hardware_concurrency(); i++)
    {

        readThreads.push_back(std::thread([&data, &exit, &shared_mtx_lock,
&readProcessedCnt]() {
            std::list<int> mydata;
            int localProcessCnt = 0;

            while (true)
            {
                shared_mtx_lock.lock_shared();

                mydata.push_back(data.back());
                ++localProcessCnt;

                shared_mtx_lock.unlock_shared();

                if (exit)
                    break;
            }

            std::atomic_fetch_add(&readProcessedCnt, localProcessCnt);

        }));

        writeThreads.push_back(std::thread([&data, &exit, &shared_mtx_lock,
&writeProcessedCnt]() {

            int localProcessCnt = 0;

            while (true)
            {
                shared_mtx_lock.lock();

                data.push_back(rand() % 100);
                ++localProcessCnt;
```

```
                shared_mtx_lock.unlock();

                if (exit)
                    break;
            }

            std::atomic_fetch_add(&writeProcessedCnt, localProcessCnt);

        }));
    }

    std::this_thread::sleep_for(std::chrono::milliseconds(MAIN_WAIT_MILLISECONDS));
    exit = true;

    for (auto &r : readThreads)
        r.join();

    for (auto &w : writeThreads)
        w.join();

    std::cout << "STLSharedTimedMutex READ :      " << readProcessedCnt << std::endl;
    std::cout << "STLSharedTimedMutex WRITE :     " << writeProcessedCnt << std::endl;
    std::cout << "TOTAL READ&WRITE :              " << readProcessedCnt + writeProcessedCnt <<
std::endl << std::endl;
}
```

Examples

std :: unique_lock, std :: shared_lock, std :: lock_guard

Wird für den RAII-Stil zum Erwerb von Try-Sperren, zeitgesteuerten Try-Sperren und rekursiven Sperren verwendet.

`std::unique_lock` erlaubt den ausschließlichen Besitz von Mutexen.

`std::shared_lock` können Mutexe gemeinsam genutzt werden. Mehrere Threads können `std::shared_locks` auf einem `std::shared_mutex` . Verfügbar ab C ++ 14.

`std::lock_guard` ist eine leichtgewichtige Alternative zu `std::unique_lock` und `std::shared_lock` .

```
#include <unordered_map>
#include <mutex>
#include <shared_mutex>
#include <thread>
#include <string>
#include <iostream>

class PhoneBook {
public:
    std::string getPhoneNo( const std::string & name )
    {
        std::shared_lock<std::shared_timed_mutex> l(_protect);
        auto it =  _phonebook.find( name );
        if ( it != _phonebook.end() )
            return (*it).second;
```

```
        return "";
    }
    void addPhoneNo ( const std::string & name, const std::string & phone )
    {
        std::unique_lock<std::shared_timed_mutex> l(_protect);
        _phonebook[name] = phone;
    }

    std::shared_timed_mutex _protect;
    std::unordered_map<std::string,std::string>  _phonebook;
};
```

Strategien für Sperrklassen: std :: try_to_lock, std :: adopt_lock, std :: defer_lock

Beim Erstellen eines std :: unique_lock stehen drei verschiedene Sperrstrategien zur Auswahl: `std::try_to_lock` , `std::defer_lock` und `std::adopt_lock`

1. `std::try_to_lock` kann eine Sperre ohne Blockierung versucht werden:

```
{
    std::atomic_int temp {0};
    std::mutex _mutex;

    std::thread t( [&](){

        while( temp!= -1){
            std::this_thread::sleep_for(std::chrono::seconds(5));
            std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);

            if(lock.owns_lock()){
                //do something
                temp=0;
            }
        }
    });

    while ( true )
    {
        std::this_thread::sleep_for(std::chrono::seconds(1));
        std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);
        if(lock.owns_lock()){
            if (temp < INT_MAX){
                ++temp;
            }
            std::cout << temp << std::endl;
        }
    }
}
```

2. `std::defer_lock` kann eine `std::defer_lock` erstellt werden, ohne die Sperre zu erhalten. Wenn mehr als ein Mutex gesperrt wird, gibt es ein Zeitfenster für einen Deadlock, wenn zwei Funktionsaufrufe gleichzeitig versuchen, die Sperren abzurufen:

```
{
    std::unique_lock<std::mutex> lock1(_mutex1, std::defer_lock);
```

```
    std::unique_lock<std::mutex> lock2(_mutex2, std::defer_lock);
    lock1.lock()
    lock2.lock(); // deadlock here
    std::cout << "Locked! << std::endl;
    //...
}
```

Mit dem folgenden Code werden die Sperren unabhängig von der Funktion in der entsprechenden Reihenfolge abgerufen und freigegeben:

```
   {
       std::unique_lock<std::mutex> lock1(_mutex1, std::defer_lock);
       std::unique_lock<std::mutex> lock2(_mutex2, std::defer_lock);
       std::lock(lock1,lock2); // no deadlock possible
       std::cout << "Locked! << std::endl;
       //...

   }
```

3. `std::adopt_lock` versucht nicht, ein zweites Mal zu sperren, wenn der aufrufende Thread derzeit die Sperre besitzt.

```
{
    std::unique_lock<std::mutex> lock1(_mutex1, std::adopt_lock);
    std::unique_lock<std::mutex> lock2(_mutex2, std::adopt_lock);
    std::cout << "Locked! << std::endl;
    //...
}
```

Dabei ist zu beachten, dass std :: adopt_lock keinen Ersatz für die rekursive Verwendung von Mutex darstellt. Wenn die Sperre den Gültigkeitsbereich verlässt, wird der Mutex **freigegeben** .

std :: mutex

std :: mutex ist eine einfache, nicht rekursive Synchronisationsstruktur, die zum Schutz von Daten verwendet wird, auf die mehrere Threads zugreifen.

```
    std::atomic_int temp{0};
    std::mutex _mutex;

    std::thread t( [&](){

                      while( temp!= -1){
                          std::this_thread::sleep_for(std::chrono::seconds(5));
                          std::unique_lock<std::mutex> lock( _mutex);

                              temp=0;
                      }
                  });


    while ( true )
    {
        std::this_thread::sleep_for(std::chrono::milliseconds(1));
        std::unique_lock<std::mutex> lock( _mutex, std::try_to_lock);
```

```
        if ( temp < INT_MAX )
            temp++;
        cout << temp << endl;

    }
```

std :: scoped_lock (C ++ 17)

`std::scoped_lock` bietet eine RAII-Semantik für den Besitz eines weiteren Mutex in Kombination mit den von `std::lock` verwendeten `std::lock` . Wenn `std::scoped_lock` zerstört wird, werden Mutexe in umgekehrter Reihenfolge freigegeben, aus der sie erworben wurden.

```
{
    std::scoped_lock lock{_mutex1,_mutex2};
    //do something
}
```

Mutex-Typen

C ++ 1x bietet eine Auswahl an Mutex-Klassen:

- std :: mutex - bietet einfache Sperrfunktionen.
- std :: timed_mutex - bietet try_to_lock-Funktionalität
- std :: recursive_mutex - Ermöglicht das rekursive Sperren durch denselben Thread.
- std :: shared_mutex, std :: shared_timed_mutex - bietet gemeinsame und einzigartige Sperrfunktionen.

std :: lock

`std::lock` verwendet Deadlock-Vermeidungsalgorithmen, um ein oder mehrere Mutexe zu sperren. Wenn während eines Aufrufs eine Ausnahme ausgelöst wird, um mehrere Objekte zu sperren, gibt `std::lock` die erfolgreich gesperrten Objekte frei, bevor die Ausnahme erneut ausgelöst wird.

```
std::lock(_mutex1, _mutex2);
```

Mutexe online lesen: https://riptutorial.com/de/cplusplus/topic/9895/mutexe

Kapitel 76: Namensräume

Einführung

Ein Namensraum ist ein deklaratives Präfix für Funktionen, Klassen, Typen usw., um Namenskollisionen bei der Verwendung mehrerer Bibliotheken zu verhindern.

Syntax

- Namespace - *Kennung (opt) {declaration-Seq}*
- Inline-Namespace- *ID* (*opt*) { *Deklaration-seq* } / * seit C ++ 11 * /
- Inline (*opt*) -Namensraum *Attribut- Bezeichner -Seq- Bezeichner* (*Opt*) { *Deklaration-Seq* } / * seit C ++ 17 * /
- Namespace *einschließender-Namespace- Bezeichner* : *ID* { *Deklaration-Seq* } / * seit C ++ 17 * /
- Namespace- *ID* = *Qualifizierter-Namespace- Bezeichner* ;
- unter Verwendung des Namespaces *verschachtelter-Name-Bezeichner* (*opt*) *Namespace-Name* ;
- *Attributspezifizierer-seq unter* Verwendung des Namespaces *verschachtelter Namensspezifizierer* (*opt*) *Namespace-Name* ; / * seit C ++ 11 * /

Bemerkungen

Das Schlüsselwort `namespace` hat drei verschiedene Bedeutungen je nach Kontext:

1. Wenn auf einen optionalen Namen und eine geschweifte Klammerfolge von Deklarationen gefolgt wird, wird ein neuer Namespace definiert oder ein vorhandener Namespace um diese Deklarationen erweitert. Wenn der Name weggelassen wird, ist der Namespace ein unbenannter Namespace .

2. Wenn ein Name und ein Gleichheitszeichen gefolgt werden, wird ein Namespace-Alias angegeben .

3. Wenn vorangestelltem `using` und von einem Namespace - Namen gefolgt, bildet es ein *using - Direktive* , die Namen im angegebenen Namespace ermöglicht durch unqualifizierte Namen - Suche gefunden werden (aber nicht die Namen im aktuellen Bereich neu deklarieren). Eine *using-Direktive* darf nicht im Klassenbereich vorkommen.

`using namespace std;` ist entmutigt. Warum? Denn der `namespace std` ist riesig! Dies bedeutet, dass die Wahrscheinlichkeit groß ist, dass Namen kollidieren:

```
//Really bad!
using namespace std;

//Calculates p^e and outputs it to std::cout
void pow(double p, double e) { /*...*/ }
```

```
//Calls pow
pow(5.0, 2.0); //Error! There is already a pow function in namespace std with the same
signature,
               //so the call is ambiguous
```

Examples

Was sind Namespaces?

Ein C ++ - Namespace ist eine Sammlung von C ++ - Entitäten (Funktionen, Klassen, Variablen), deren Namen den Namen des Namespaces voranstellen. Beim Schreiben von Code innerhalb eines Namespaces müssen benannten Entitäten, die zu diesem Namespace gehören, nicht der Name des Namespaces vorangestellt werden. Entitäten außerhalb dieses Namespaces müssen jedoch den vollständig qualifizierten Namen verwenden. Der vollständig qualifizierte Name hat das Format `<namespace>::<entity>` . Beispiel:

```
namespace Example
{
  const int test = 5;

  const int test2 = test + 12; //Works within `Example` namespace
}

const int test3 = test + 3; //Fails; `test` not found outside of namespace.

const int test3 = Example::test + 3; //Works; fully qualified name used.
```

Namespaces sind nützlich, um zusammengehörige Definitionen zu gruppieren. Nehmen Sie die Analogie eines Einkaufszentrums. In der Regel ist ein Einkaufszentrum in mehrere Geschäfte aufgeteilt, von denen jeder Artikel aus einer bestimmten Kategorie verkauft. Ein Geschäft kann Elektronik verkaufen, während ein anderes Geschäft Schuhe verkaufen kann. Diese logischen Trennungen in den Geschäftstypen helfen den Käufern, die Artikel zu finden, nach denen sie suchen. Namespaces helfen C ++ - Programmierern wie Käufern dabei, die gewünschten Funktionen, Klassen und Variablen zu finden, indem sie auf logische Weise organisiert werden. Beispiel:

```
namespace Electronics
{
    int TotalStock;
    class Headphones
    {
        // Description of a Headphone (color, brand, model number, etc.)
    };
    class Television
    {
        // Description of a Television (color, brand, model number, etc.)
    };
}

namespace Shoes
{
```

```
    int TotalStock;
    class Sandal
    {
        // Description of a Sandal (color, brand, model number, etc.)
    };
    class Slipper
    {
        // Description of a Slipper (color, brand, model number, etc.)
    };
}
```

Es ist ein einzelner Namespace vordefiniert. Hierbei handelt es sich um den globalen Namespace, der keinen Namen hat, aber mit `::` bezeichnet werden kann. Beispiel:

```
void bar() {
    // defined in global namespace
}
namespace foo {
    void bar() {
        // defined in namespace foo
    }
    void barbar() {
        bar();   // calls foo::bar()
        ::bar(); // calls bar() defined in global namespace
    }
}
```

Namensräume erstellen

Das Erstellen eines Namespaces ist sehr einfach:

```
//Creates namespace foo
namespace Foo
{
    //Declares function bar in namespace foo
    void bar() {}
}
```

So rufen Sie `bar` , müssen Sie zuerst den Namespace angeben, gefolgt von dem Bereichsauflösungsoperator `::` :

```
Foo::bar();
```

Es ist erlaubt, einen Namespace in einem anderen zu erstellen, zum Beispiel:

```
namespace A
{
    namespace B
    {
        namespace C
        {
            void bar() {}
        }
    }
```

```
}
```

C ++ 17

Der obige Code könnte folgendermaßen vereinfacht werden:

```
namespace A::B::C
{
    void bar() {}
}
```

Namensräume erweitern

Eine nützliche Funktion von `namespace` ist, dass Sie sie erweitern (Mitglieder hinzufügen) können.

```
namespace Foo
{
    void bar() {}
}

//some other stuff

namespace Foo
{
    void bar2() {}
}
```

Direktive verwenden

Das Schlüsselwort "using" hat drei Varianten. Zusammen mit dem Schlüsselwort 'Namespace' schreiben Sie eine 'using-Direktive':

Wenn Sie nicht `Foo::` vor jedem Zeugs im Namespace `Foo` schreiben möchten, können Sie den `using namespace Foo;` alles aus `Foo` zu importieren.

```
namespace Foo
{
    void bar() {}
    void baz() {}
}

//Have to use Foo::bar()
Foo::bar();

//Import Foo
using namespace Foo;
bar(); //OK
baz(); //OK
```

Es ist auch möglich, ausgewählte Entitäten in einem Namensraum und nicht im gesamten Namensraum zu importieren:

```
using Foo::bar;
```

```
bar(); //OK, was specifically imported
baz(); // Not OK, was not imported
```

Ein Wort der Vorsicht: Die `using namespace` in Header-Dateien wird in den meisten Fällen als fehlerhaft angesehen. In diesem Fall wird der Namespace in *jede* Datei importiert, die den Header enthält. Da gibt es keine Möglichkeit der „un- `using` “ einem Namespace, dies zu Namespace Verschmutzung führen kann (mehr oder unerwarteter Symbole im globalen Namespace) oder, schlimmer noch, Konflikte. In diesem Beispiel wird das Problem veranschaulicht:

```
/***** foo.h *****/
namespace Foo
{
    class C;
}

/***** bar.h *****/
namespace Bar
{
    class C;
}

/***** baz.h *****/
#include "foo.h"
using namespace Foo;

/***** main.cpp *****/
#include "bar.h"
#include "baz.h"

using namespace Bar;
C c; // error: Ambiguity between Bar::C and Foo::C
```

Eine *using-Direktive* darf nicht im Klassenbereich vorkommen.

Argumentabhängige Suche

Beim Aufrufen einer Funktion ohne expliziten Namespace-Qualifikationsmerkmal kann der Compiler eine Funktion innerhalb eines Namespaces aufrufen, wenn sich einer der Parametertypen für diese Funktion ebenfalls in diesem Namespace befindet. Dies wird als "Argumentabhängige Suche" oder ADL bezeichnet:

```
namespace Test
{
  int call(int i);

  class SomeClass {...};

  int call_too(const SomeClass &data);
}

call(5); //Fails. Not a qualified function name.

Test::SomeClass data;

call_too(data); //Succeeds
```

`call` schlägt fehl, da keiner seiner Parametertypen aus dem `Test` Namespace stammt. `call_too` funktioniert, weil `SomeClass` Mitglied von `Test` und sich daher für ADL-Regeln qualifiziert.

Wann tritt ADL nicht auf

ADL tritt nicht auf, wenn die normale unqualifizierte Suche nach einem Klassenmitglied, einer im Blockbereich deklarierten Funktion oder einem nicht funktionellen Typ sucht. Zum Beispiel:

```
void foo();
namespace N {
    struct X {};
    void foo(X ) { std::cout << '1'; }
    void qux(X ) { std::cout << '2'; }
}

struct C {
    void foo() {}
    void bar() {
        foo(N::X{}); // error: ADL is disabled and C::foo() takes no arguments
    }
};

void bar() {
    extern void foo(); // redeclares ::foo
    foo(N::X{});       // error: ADL is disabled and ::foo() doesn't take any arguments
}

int qux;

void baz() {
    qux(N::X{}); // error: variable declaration disables ADL for "qux"
}
```

Inline-Namespace

C ++ 11

`inline namespace` enthält den Inhalt des Inline-Namespaces im umschließenden Namespace, so

```
namespace Outer
{
    inline namespace Inner
    {
        void foo();
    }
}
```

ist meistens äquivalent zu

```
namespace Outer
{

    namespace Inner
    {
        void foo();
```

```
    }

    using Inner::foo;
}
```

Elemente aus `Outer::Inner::` und denen, die `Outer::` sind, sind jedoch identisch.

Folgendes ist also gleichwertig

```
Outer::foo();
Outer::Inner::foo();
```

Die Alternative `using namespace Inner;` wäre für einige knifflige Teile als Schablonenspezialisierung nicht gleichwertig:

Zum

```
#include <outer.h> // See below

class MyCustomType;
namespace Outer
{
    template <>
    void foo<MyCustomType>() { std::cout << "Specialization"; }
}
```

- Der Inline-Namespace ermöglicht die Spezialisierung von `Outer::foo`

  ```
  // outer.h
  // include guard omitted for simplification

  namespace Outer
  {
      inline namespace Inner
      {
          template <typename T>
          void foo() { std::cout << "Generic"; }
      }
  }
  ```

- Während der `using namespace` die Spezialisierung von `Outer::foo` nicht zulässt

  ```
  // outer.h
  // include guard omitted for simplification

  namespace Outer
  {
      namespace Inner
      {
          template <typename T>
          void foo() { std::cout << "Generic"; }
      }
      using namespace Inner;
      // Specialization of `Outer::foo` is not possible
      // it should be `Outer::Inner::foo`.
  ```

```
}
```

Inline-Namespace ist eine Möglichkeit, mehreren Versionen das Zusammenleben und die Verwendung der `inline` Version zu ermöglichen

```
namespace MyNamespace
{
    // Inline the last version
    inline namespace Version2
    {
        void foo(); // New version
        void bar();
    }

    namespace Version1 // The old one
    {
        void foo();
    }

}
```

Und mit der Nutzung

```
MyNamespace::Version1::foo(); // old version
MyNamespace::Version2::foo(); // new version
MyNamespace::foo();           // default version : MyNamespace::Version1::foo();
```

Unbenannte / anonyme Namespaces

Ein unbenannter Namespace kann verwendet werden, um sicherzustellen, dass Namen über eine interne Verknüpfung verfügen (auf die nur die aktuelle Übersetzungseinheit verweist). Ein solcher Namespace wird wie jeder andere Namespace definiert, jedoch ohne den Namen:

```
namespace {
    int foo = 42;
}
```

`foo` ist nur in der Übersetzungseinheit sichtbar, in der es angezeigt wird.

Es wird empfohlen, niemals unbenannte Namespaces in Header-Dateien zu verwenden, da hierdurch für jede Übersetzungseinheit, in der sie enthalten ist, eine Version des Inhalts angezeigt wird. Dies ist besonders wichtig, wenn Sie Nicht-Konstante-Globals definieren.

```
// foo.h
namespace {
    std::string globalString;
}

// 1.cpp
#include "foo.h" //< Generates unnamed_namespace{1.cpp}::globalString ...

globalString = "Initialize";
```

```
// 2.cpp
#include "foo.h" //< Generates unnamed_namespace{2.cpp}::globalString ...

std::cout << globalString; //< Will always print the empty string
```

Kompakte verschachtelte Namespaces

C ++ 17

```
namespace a {
  namespace b {
    template<class T>
    struct qualifies : std::false_type {};
  }
}

namespace other {
  struct bob {};
}

namespace a::b {
  template<>
  struct qualifies<::other::bob> : std::true_type {};
}
```

Sie können sowohl die Eingabe `a` und `b` Namespaces in einem Schritt mit `namespace a::b` beginnend in C ++ 17.

Aliasing eines langen Namespaces

Dies wird normalerweise zum Umbenennen oder Verkürzen langer Namespace-Referenzen verwendet, z.

```
namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}

namespace Name1 = boost::multiprecision;


//    Both Type declarations are equivalent
boost::multiprecision::Number X   //    Writing the full namespace path, longer
Name1::Number Y                   //    using the name alias, shorter
```

Alias-Deklarationsumfang

Alias Erklärung werden durch *Verwendung* der vorstehenden Aussagen betroffen

```
namespace boost
{
```

```
    namespace multiprecision
    {
        class Number ...
    }
}


using namespace boost;

//   Both Namespace are equivalent
namespace Name1 = boost::multiprecision;
namespace Name2 = multiprecision;
```

Es ist jedoch einfacher zu verwechseln, welchen Namespace Sie als Aliasing verwenden, wenn Sie Folgendes haben:

```
namespace boost
{
    namespace multiprecision
    {
        class Number ...
    }
}

namespace numeric
{
    namespace multiprecision
    {
        class Number ...
    }
}

using namespace numeric;
using namespace boost;

//    Not recommended as
//    its not explicitly clear whether Name1 refers to
//    numeric::multiprecision or boost::multiprecision
namespace Name1 = multiprecision;

//    For clarity, its recommended to use absolute paths
//    instead
namespace Name2 = numeric::multiprecision;
namespace Name3 = boost::multiprecision;
```

Namespace-Alias

Einem Namespace kann ein Alias (*dh ein* anderer Name für denselben Namespace) mithilfe der `namespace` *ID bezeichner* `=` zugewiesen werden. Auf Mitglieder des Alias-Namespaces kann zugegriffen werden, indem sie mit dem Namen des Alias qualifiziert werden. Im folgenden Beispiel ist der geschachtelte Namespace `AReallyLongName::AnotherReallyLongName` ungünstig, sodass die Funktion `qux` lokal einen Alias `N` deklariert. Auf Mitglieder dieses Namensraums kann dann einfach mit `N::` zugegriffen werden.

```
namespace AReallyLongName {
    namespace AnotherReallyLongName {
```

```
        int foo();
        int bar();
        void baz(int x, int y);
    }
}
void qux() {
    namespace N = AReallyLongName::AnotherReallyLongName;
    N::baz(N::foo(), N::bar());
}
```

Namensräume online lesen: https://riptutorial.com/de/cplusplus/topic/495/namensraume

Kapitel 77: Neugierig wiederkehrendes Vorlagenmuster (CRTP)

Einführung

Ein Muster, in dem eine Klasse von einer Klassenvorlage mit sich selbst als einem ihrer Vorlagenparameter erbt. CRTP wird normalerweise verwendet, um *statischen Polymorphismus* in C ++ bereitzustellen.

Examples

Das kurioserweise wiederkehrende Vorlagenmuster (CRTP)

CRTP ist eine leistungsstarke, statische Alternative zu virtuellen Funktionen und herkömmlicher Vererbung, mit der Typeneigenschaften zur Kompilierzeit angegeben werden können. Es verfügt über eine Basisklassenvorlage, die die abgeleitete Klasse als einen ihrer Vorlagenparameter übernimmt. Dies erlaubt es ihm, legal einen `static_cast` seines `this` Zeigers auf die abgeleitete Klasse durchzuführen.

Das bedeutet natürlich auch, dass eine CRTP-Klasse *immer* als Basisklasse einer anderen Klasse verwendet werden muss. Und die abgeleitete Klasse muss sich selbst an die Basisklasse übergeben.

C ++ 14

Nehmen wir an, Sie haben eine Reihe von Containern, die alle die Funktionen `begin()` und `end()` . Die Anforderungen der Standardbibliothek für Container erfordern mehr Funktionalität. Wir können eine CRTP-Basisklasse entwerfen, die diese Funktionalität ausschließlich auf der Grundlage von `begin()` und `end()` bereitstellt:

```
#include <iterator>
template <typename Sub>
class Container {
  private:
    // self() yields a reference to the derived type
    Sub& self() { return *static_cast<Sub*>(this); }
    Sub const& self() const { return *static_cast<Sub const*>(this); }

  public:
    decltype(auto) front() {
      return *self().begin();
    }

    decltype(auto) back() {
      return *std::prev(self().end());
    }

    decltype(auto) size() const {
```

```
      return std::distance(self().begin(), self().end());
    }

    decltype(auto) operator[](std::size_t i) {
      return *std::next(self().begin(), i);
    }
};
```

Die obige Klasse stellt die Funktionen `front()` , `back()` , `size()` und `operator[]` für jede Unterklasse bereit, die `begin()` und `end()` bereitstellt. Eine beispielhafte Unterklasse ist ein einfaches, dynamisch zugewiesenes Array:

```
#include <memory>
// A dynamically allocated array
template <typename T>
class DynArray : public Container<DynArray<T>> {
  public:
    using Base = Container<DynArray<T>>;

    DynArray(std::size_t size)
      : size_{size},
      data_{std::make_unique<T[]>(size_)}
    { }

    T* begin() { return data_.get(); }
    const T* begin() const { return data_.get(); }
    T* end() { return data_.get() + size_; }
    const T* end() const { return data_.get() + size_; }

  private:
    std::size_t size_;
    std::unique_ptr<T[]> data_;
};
```

Benutzer der `DynArray` Klasse können die von der CRTP-Basisklasse bereitgestellten Schnittstellen problemlos wie folgt verwenden:

```
DynArray<int> arr(10);
arr.front() = 2;
arr[2] = 5;
assert(arr.size() == 10);
```

Nützlichkeit: Dieses Muster vermeidet insbesondere virtuelle Funktionsaufrufe zur Laufzeit, die zum Durchlaufen der Vererbungshierarchie auftreten und stützen sich einfach auf statische Umsetzungen:

```
DynArray<int> arr(10);
DynArray<int>::Base & base = arr;
base.begin(); // no virtual calls
```

Die einzige statische `Container<DynArray<int>>` innerhalb der Funktion `begin()` in der Basisklasse `Container<DynArray<int>>` ermöglicht es dem Compiler, den Code drastisch zu optimieren, und zur Laufzeit erfolgt keine virtuelle Tabellensuche.

Einschränkungen: Da die Basisklasse für zwei verschiedene `DynArray` Klassen unterschiedlich `DynArray` , ist es nicht möglich, Zeiger auf ihre Basisklassen in einem typhomogenen Array zu speichern, wie dies bei normaler Vererbung der `DynArray` wäre, bei der die Basisklasse nicht von der abgeleiteten Klasse abhängt Art:

```
class A {};
class B: public A{};

A* a = new B;
```

CRTP, um Code-Duplizierung zu vermeiden

Das Beispiel in Visitor Pattern bietet einen überzeugenden Anwendungsfall für CRTP:

```
struct IShape
{
    virtual ~IShape() = default;

    virtual void accept(IShapeVisitor&) const = 0;
};

struct Circle : IShape
{
    // ...
    // Each shape has to implement this method the same way
    void accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }
    // ...
};

struct Square : IShape
{
    // ...
    // Each shape has to implement this method the same way
    void accept(IShapeVisitor& visitor) const override { visitor.visit(*this); }
    // ...
};
```

Jeder `IShape` Typ von `IShape` muss dieselbe Funktion auf dieselbe Weise implementieren. Das ist viel mehr Tippen. Stattdessen können wir einen neuen Typ in der Hierarchie einführen, der dies für uns tut:

```
template <class Derived>
struct IShapeAcceptor : IShape {
    void accept(IShapeVisitor& visitor) const override {
        // visit with our exact type
        visitor.visit(*static_cast<Derived const*>(this));
    }
};
```

Und jetzt muss jede Form einfach vom Akzeptor erben:

```
struct Circle : IShapeAcceptor<Circle>
{
    Circle(const Point& center, double radius) : center(center), radius(radius) {}
```

```
    Point center;
    double radius;
};

struct Square : IShapeAcceptor<Square>
{
    Square(const Point& topLeft, double sideLength) : topLeft(topLeft), sideLength(sideLength)
{}
    Point topLeft;
    double sideLength;
};
```

Kein doppelter Code erforderlich.

Neugierig wiederkehrendes Vorlagenmuster (CRTP) online lesen:
https://riptutorial.com/de/cplusplus/topic/9269/neugierig-wiederkehrendes-vorlagenmuster--crtp-

Kapitel 78: Nicht statische Memberfunktionen

Syntax

- // Anrufen:
    - Variable.Mitgliedsfunktion ();
    - variable_pointer-> member_function ();
- // Definition:
    - ret_type klassenname :: member_function () cv-qualifiers {
        - Karosserie;
    - }
- // Prototyp:
    - class class_name {
        - virt-specifier ret_type member_function () cv-qualifiers virt-spezifier-seq;
        - // virt-spezifier: "virtuell", falls zutreffend.
        - // cv-qualifiers: "const" und / oder "volatile", falls zutreffend.
        - // virt-specifier-seq: "override" und / oder "final", falls zutreffend.
    - }

Bemerkungen

Eine nicht `static` Mitgliedsfunktion ist eine `class` / `struct` / `union` , die für eine bestimmte Instanz aufgerufen wird und für diese Instanz arbeitet. Im Gegensatz zu `static` Memberfunktionen kann sie nicht ohne Angabe einer Instanz aufgerufen werden.

Informationen zu Klassen, Strukturen und Vereinigungen finden Sie im übergeordneten Thema .

Examples

Nicht statische Elementfunktionen

Eine `class` oder `struct` kann sowohl Member-Funktionen als auch Member-Variablen haben. Diese Funktionen haben eine weitgehend eigenständige Syntax und können entweder innerhalb oder außerhalb der Klassendefinition definiert werden. Wenn die Funktion außerhalb der Klassendefinition definiert ist, wird dem Namen der Funktion der Name der Klasse und der Gültigkeitsbereich-Operator (`::` :) vorangestellt.

```
class CL {
  public:
    void  definedInside() {}
    void definedOutside();
```

```
};
void CL::definedOutside() {}
```

Diese Funktionen werden für eine Instanz (oder einen Verweis auf eine Instanz) der Klasse mit dem Punkt (`.`) Oder einem Zeiger auf eine Instanz mit dem Pfeil (`->`) aufgerufen. Jeder Aufruf ist an die Instanz der Funktion gebunden wurde angerufen; Wenn eine Member-Funktion für eine Instanz aufgerufen wird, hat sie Zugriff auf alle Felder dieser Instanz (über `this` Zeiger), kann jedoch nur auf die Felder anderer Instanzen zugreifen, wenn diese Instanzen als Parameter angegeben werden.

```
struct ST {
    ST(const std::string& ss = "Wolf", int ii = 359) : s(ss), i(ii) { }

    int get_i() const { return i; }
    bool compare_i(const ST& other) const { return (i == other.i); }

  private:
    std::string s;
    int i;
};
ST st1;
ST st2("Species", 8472);

int  i = st1.get_i(); // Can access st1.i, but not st2.i.
bool b = st1.compare_i(st2); // Can access st1 & st2.
```

Diese Funktionen können auf Member-Variablen und / oder andere Member-Funktionen zugreifen, unabhängig von den Zugriffsmodifizierern der Variablen oder Funktionen. Sie können auch außerhalb der Reihenfolge geschrieben werden, um auf Member-Variablen und / oder aufrufende Member-Funktionen zuzugreifen, die vor ihnen deklariert wurden, da die gesamte Klassendefinition analysiert werden muss, bevor der Compiler mit der Kompilierung einer Klasse beginnen kann.

```
class Access {
  public:
    Access(int i_ = 8088, int j_ = 8086, int k_ = 6502) : i(i_), j(j_), k(k_) {}

    int i;
    int get_k() const { return k; }
    bool private_no_more() const { return i_be_private(); }
  protected:
    int j;
    int get_i() const { return i; }
  private:
    int k;
    int get_j() const { return j; }
    bool i_be_private() const { return ((i > j) && (k < j)); }
};
```

Verkapselung

Member-Funktionen werden häufig für die Verkapselung verwendet. Sie verwenden einen *Accessor* (allgemein als Getter bezeichnet) und einen *Mutator* (allgemein als Setter bezeichnet),

anstatt direkt auf Felder zuzugreifen.

```
class Encapsulator {
    int encapsulated;

  public:
    int  get_encapsulated() const { return encapsulated; }
    void set_encapsulated(int e)  { encapsulated = e; }

    void some_func() {
        do_something_with(encapsulated);
    }
};
```

Innerhalb der Klasse kann `encapsulated` von jeder nicht statischen `encapsulated` frei zugegriffen werden. Außerhalb der Klasse wird der Zugriff darauf durch Member-Funktionen geregelt. Verwenden Sie `get_encapsulated()` , um sie zu lesen, und `set_encapsulated()` , um sie zu ändern. Dies verhindert unbeabsichtigte Änderungen an der Variablen, da zum Lesen und Schreiben separate Funktionen verwendet werden. [Es gibt viele Diskussionen darüber, ob Getter und Setter die Einkapselung anbieten oder brechen, mit guten Argumenten für beide Ansprüche; Eine solche hitzige Debatte fällt nicht in den Rahmen dieses Beispiels.]

Name ausblenden & importieren

Wenn eine Basisklasse einen Satz überladener Funktionen bereitstellt und eine abgeleitete Klasse dem Satz eine weitere Überladung hinzufügt, werden alle von der Basisklasse bereitgestellten Überladungen ausgeblendet.

```
struct HiddenBase {
    void f(int) { std::cout << "int" << std::endl; }
    void f(bool) { std::cout << "bool" << std::endl; }
    void f(std::string) { std::cout << "std::string" << std::endl; }
};

struct HidingDerived : HiddenBase {
    void f(float) { std::cout << "float" << std::endl; }
};

// ...

HiddenBase hb;
HidingDerived hd;
std::string s;

hb.f(1);    // Output:  int
hb.f(true); // Output:  bool
hb.f(s);    // Output:  std::string;

hd.f(1.f);  // Output:  float
hd.f(3);    // Output:  float
hd.f(true); // Output:  float
hd.f(s);    // Error: Can't convert from std::string to float.
```

Dies ist auf Namensauflösungsregeln zurückzuführen: Während der Namenssuche, sobald der richtige Name gefunden wurde, hören wir auf zu suchen, selbst wenn wir eindeutig nicht die

richtige *Version* der Entität mit diesem Namen gefunden haben (z. B. mit `hd.f(s)`); Aus diesem Grund verhindert das Überladen der Funktion in der abgeleiteten Klasse, dass die Namenssuche die Überladungen in der Basisklasse erkennt. Um dies zu vermeiden, können mithilfe einer using-Deklaration Namen aus der Basisklasse in die abgeleitete Klasse "importiert" werden, sodass sie während der Namenssuche verfügbar sind.

```
struct HidingDerived : HiddenBase {
     // All members named HiddenBase::f shall be considered members of HidingDerived for
lookup.
    using HiddenBase::f;

    void f(float) { std::cout << "float" << std::endl; }
};

// ...

HidingDerived hd;

hd.f(1.f);  // Output:  float
hd.f(3);    // Output:  int
hd.f(true); // Output:  bool
hd.f(s);    // Output:  std::string
```

Wenn eine abgeleitete Klasse Namen mit einer using-Deklaration importiert, aber Funktionen mit derselben Signatur wie Funktionen in der Basisklasse deklariert, werden die Funktionen der Basisklasse unbemerkt überschrieben oder ausgeblendet.

```
struct NamesHidden {
    virtual void hide_me()      {}
    virtual void hide_me(float) {}
    void hide_me(int)           {}
    void hide_me(bool)          {}
};

struct NameHider : NamesHidden {
    using NamesHidden::hide_me;

    void hide_me()    {} // Overrides NamesHidden::hide_me().
    void hide_me(int) {} // Hides NamesHidden::hide_me(int).
};
```

Eine using-Deklaration kann auch verwendet werden, um Zugriffsmodifizierer zu ändern, vorausgesetzt, die importierte Entität war `public` oder in der Basisklasse `protected` .

```
struct ProMem {
  protected:
    void func() {}
};

struct BecomesPub : ProMem {
    using ProMem::func;
};

// ...

ProMem pm;
```

```
BecomesPub bp;

pm.func(); // Error: protected.
bp.func(); // Good.
```

Wenn wir explizit eine Member-Funktion von einer bestimmten Klasse in der Vererbungshierarchie aufrufen möchten, können wir den Funktionsnamen beim Aufruf der Funktion angeben, indem Sie diese Klasse nach Namen angeben.

```
struct One {
    virtual void f() { std::cout << "One." << std::endl; }
};

struct Two : One {
    void f() override {
        One::f(); // this->One::f();
        std::cout << "Two." << std::endl;
    }
};

struct Three : Two {
    void f() override {
        Two::f(); // this->Two::f();
        std::cout << "Three." << std::endl;
    }
};

// ...

Three t;

t.f();      // Normal syntax.
t.Two::f(); // Calls version of f() defined in Two.
t.One::f(); // Calls version of f() defined in One.
```

Virtuelle Elementfunktionen

Elementfunktionen können auch als `virtual` deklariert werden. Wenn in diesem Fall ein Zeiger oder eine Referenz auf eine Instanz aufgerufen wird, wird nicht direkt auf sie zugegriffen. `vtable` suchen sie die Funktion in der virtuellen Funktionstabelle (eine Liste von Zeiger-zu-Member-Funktionen für virtuelle Funktionen, die üblicherweise als `vtable` oder `vftable`) und verwenden sie, um die für die Dynamik der Instanz geeignete Version aufzurufen (tatsächlicher) Typ. Wenn die Funktion direkt aus einer Variablen einer Klasse aufgerufen wird, wird keine Suche durchgeführt.

```
struct Base {
    virtual void func() { std::cout << "In Base." << std::endl; }
};

struct Derived : Base {
    void func() override { std::cout << "In Derived." << std::endl; }
};

void slicer(Base x) { x.func(); }

// ...
```

```
Base b;
Derived d;

Base *pb = &b, *pd = &d; // Pointers.
Base &rb = b, &rd = d;   // References.

b.func();   // Output:  In Base.
d.func();   // Output:  In Derived.

pb->func(); // Output:  In Base.
pd->func(); // Output:  In Derived.

rb.func();  // Output:  In Base.
rd.func();  // Output:  In Derived.

slicer(b);  // Output:  In Base.
slicer(d);  // Output:  In Base.
```

Beachten Sie, dass während `pd` `Base*` und `rd` eine `Base&` , der Aufruf von `func()` bei einem der beiden Aufrufe `Derived::func()` anstelle von `Base::func()` ; Dies liegt daran, dass die `vtable` für `Derived` den Eintrag `Base::func()` aktualisiert und stattdessen auf `Derived::func()` . Beachten Sie umgekehrt, dass das Übergeben einer Instanz an `slicer()` immer dazu führt, dass `Base::func()` aufgerufen wird, auch wenn die übergebene Instanz ein `Derived` . dies ist wegen etwas , als *Daten - Slicing* bekannt, wo eine vorübergehenden `Derived` Instanz in eine `Base` von Wertparametern den Teil der macht `Derived` Instanz , die sie nicht um eine `Base` - Instanz nicht zugegriffen werden .

Wenn eine Member-Funktion als virtuell definiert wird, überschreiben alle abgeleiteten Klassen-Member-Funktionen mit derselben Signatur diese Funktion, unabhängig davon, ob die überschreibende Funktion als `virtual` oder nicht. Dies kann dazu führen, dass abgeleitete Klassen für Programmierer schwieriger zu analysieren sind, da es keine Hinweise darauf gibt, welche Funktion (en) `virtual` .

```
struct B {
    virtual void f() {}
};

struct D : B {
    void f() {} // Implicitly virtual, overrides B::f.
                //  You'd have to check B to know that, though.
};
```

Beachten Sie jedoch, dass eine abgeleitete Funktion eine Basisfunktion nur überschreibt, wenn ihre Signaturen übereinstimmen. Selbst wenn eine abgeleitete Funktion explizit als `virtual` deklariert wird, erstellt sie stattdessen eine neue virtuelle Funktion, wenn die Signaturen nicht übereinstimmen.

```
struct BadB {
    virtual void f() {}
};

struct BadD : BadB {
    virtual void f(int i) {} // Does NOT override BadB::f.
};
```

C ++ 11

Mit C ++ 11 kann die Absicht zum Überschreiben explizit mit dem kontextsensitiven Schlüsselwort `override` . Dies teilt dem Compiler mit, dass der Programmierer erwartet, dass er eine Basisklassenfunktion überschreibt, wodurch der Compiler einen Fehler auslässt, wenn er *nichts* außer Kraft setzt.

```
struct CPP11B {
    virtual void f() {}
};

struct CPP11D : CPP11B {
    void f() override {}
    void f(int i) override {} // Error: Doesn't actually override anything.
};
```

Dies hat auch den Vorteil, den Programmierern mitzuteilen, dass die Funktion sowohl virtuell ist als auch in mindestens einer Basisklasse deklariert ist, wodurch komplexe Klassen einfacher zu analysieren sind.

Wenn eine Funktion als `virtual` deklariert und außerhalb der Klassendefinition definiert wird, muss der `virtual` Bezeichner in der Funktionsdeklaration enthalten sein und nicht in der Definition wiederholt werden.

C ++ 11

Dies gilt auch für das `override` .

```
struct VB {
    virtual void f(); // "virtual" goes here.
    void g();
};
/* virtual */ void VB::f() {} // Not here.
virtual void VB::g() {} // Error.
```

Wenn eine Basisklasse eine `virtual` Funktion überlastet, werden nur `virtual` Überladungen, die explizit als `virtual` werden, verwendet.

```
struct BOverload {
    virtual void func() {}
    void func(int) {}
};

struct DOverload : BOverload {
    void func() override {}
    void func(int) {}
};

// ...

BOverload* bo = new DOverload;
bo->func(); // Calls DOverload::func().
bo->func(1); // Calls BOverload::func(int).
```

Weitere Informationen finden Sie im entsprechenden Thema .

Const Korrektheit

Eine der Hauptanwendungen für `this` cv-Qualifier ist die *`const` Richtigkeit* . Das ist die Praxis zu gewährleisten , dass nur dieses *Bedürfnis* greift auf ein Objekt in der *Lage* , zu modifizieren , um das Objekt, und dass jedes (Mitglied oder Nicht-Mitglied) Funktion, braucht nicht zu ändern ist , ein Objekt zu modifizieren keinen Schreibzugriff auf das hat Objekt (direkt oder indirekt). Dies verhindert unbeabsichtigte Änderungen und macht Code weniger fehleranfällig. Es ermöglicht auch jede Funktion, die den Status nicht ändern muss, um entweder ein `const` oder ein non- `const` Objekt zu übernehmen, ohne die Funktion neu schreiben oder überladen zu müssen.

`const` Korrektheit beginnt naturgemäß von unten nach oben: Jede Klassen-Member-Funktion, die den Status nicht ändern muss, wird als `const` deklariert , sodass sie für `const` Instanzen aufgerufen werden kann. Auf diese Weise können übergebene Parameter als `const` deklariert werden, wenn sie nicht geändert werden müssen. Dadurch können Funktionen entweder `const` oder non- `const` Objekte aufnehmen, ohne sich zu beschweren, und `const` -ness kann sich nach außen ausbreiten Weise. Getter sind daher häufig `const` , ebenso wie alle anderen Funktionen, die den logischen Zustand nicht ändern müssen.

```
class ConstIncorrect {
    Field fld;

  public:
    ConstIncorrect(const Field& f) : fld(f) {}     // Modifies.

    const Field& get_field()       { return fld; } // Doesn't modify; should be const.
    void set_field(const Field& f) { fld = f; }    // Modifies.

    void do_something(int i) {                     // Modifies.
        fld.insert_value(i);
    }
    void do_nothing()        { }                   // Doesn't modify; should be const.
};

class ConstCorrect {
    Field fld;

  public:
    ConstCorrect(const Field& f) : fld(f) {}       // Not const: Modifies.

    const Field& get_field() const { return fld; } // const: Doesn't modify.
    void set_field(const Field& f) { fld = f; }    // Not const: Modifies.

    void do_something(int i) {                     // Not const: Modifies.
        fld.insert_value(i);
    }
    void do_nothing() const  { }                   // const: Doesn't modify.
};

// ...

const ConstIncorrect i_cant_do_anything(make_me_a_field());
// Now, let's read it...
Field f = i_cant_do_anything.get_field();
```

```
  // Error: Loses cv-qualifiers, get_field() isn't const.
i_cant_do_anything.do_nothing();
  // Error: Same as above.
// Oops.

const ConstCorrect but_i_can(make_me_a_field());
// Now, let's read it...
Field f = but_i_can.get_field(); // Good.
but_i_can.do_nothing();          // Good.
```

Wie aus den Kommentaren zu `ConstIncorrect` und `ConstCorrect` , dienen die richtigen Qualifizierungsfunktionen auch der Dokumentation. Wenn eine Klasse `const` korrekt ist, kann davon ausgegangen werden, dass jede Funktion, die nicht `const` ist, den Status ändert, und es kann davon ausgegangen werden, dass jede Funktion, die `const` ist, den Status nicht ändert.

Nicht statische Memberfunktionen online lesen:
https://riptutorial.com/de/cplusplus/topic/5661/nicht-statische-memberfunktionen

Kapitel 79: Operator Vorrang

Bemerkungen

Operatoren werden von oben nach unten in absteigender Reihenfolge aufgeführt. Operatoren mit derselben Nummer haben gleiche Priorität und dieselbe Assoziativität.

1. `::`
2. Die Postfix-Operatoren: `[] () T(...) . -> ++ -- dynamic_cast static_cast reinterpret_cast const_cast typeid`
3. Die unären Präfixoperatoren: `++ -- * & + - ! ~ sizeof new delete delete[]` ; die Cast-Notation im C-Stil `(T)...` ; (C ++ 11 und höher) `sizeof... alignof noexcept`
4. `.*` und `->*`
5. `*` , `/` und `%` , binäre arithmetische Operatoren
6. `+` und `-` , binäre arithmetische Operatoren
7. `<<` und `>>`
8. `<` , `>` , `<=` , `>=`
9. `==` und `!=`
10. `&` , der bitweise AND-Operator
11. `^`
12. `|`
13. `&&`
14. `||`
15. `?:` (ternärer bedingter Operator)
16. `=` , `*=` , `/=` , `%=` , `+=` , `-=` , `>>=` , `<<=` , `&=` , `^=` , `|=`
17. `throw`
18. `,` (Der Komma - Operator)

Die Zuweisung, die zusammengesetzte Zuweisung und die ternären bedingten Operatoren sind rechtsassoziativ. Alle anderen binären Operatoren sind linksassoziativ.

Die Regeln für den ternären Bedingungsoperator sind etwas komplizierter, als dies bei einfachen Vorrangregeln möglich ist.

- Ein Operand bindet weniger eng an ein `?` auf der linken Seite oder a `:` auf der rechten Seite als für jeden anderen Betreiber. Der zweite Operand des Bedingungsoperators wird so analysiert, als wäre er in Klammern. Dies ermöglicht einen Ausdruck wie `a ? b , c : d` um syntaktisch gültig zu sein.
- Ein Operand bindet fester an ein `?` auf der rechten Seite als auf einen Zuweisungsoperator oder `throw` auf die linke Seite, also `a = b ? c : d` ist äquivalent zu `a = (b ? c : d)` und `throw a ? b : c` ist gleichbedeutend mit `throw (a ? b : c)` .
- Ein Operand bindet fester an einen Zuweisungsoperator an seiner rechten Seite als an `:` zu seiner Linken, also `a ? b : c = d` ist äquivalent zu `a ? b : (c = d)` .

Examples

Rechenzeichen

Arithmetische Operatoren in C ++ haben die gleiche Priorität wie in der Mathematik:

Multiplikation und Division haben Assoziativität hinterlassen (was bedeutet, dass sie von links nach rechts ausgewertet werden) und sie haben eine höhere Priorität als Addition und Subtraktion, die auch Assoziativität hinterlassen haben.

Wir können den Vorrang des Ausdrucks auch mit Klammern `( )` erzwingen. Genauso wie Sie das in der normalen Mathematik tun würden.

```
// volume of a spherical shell = 4 pi R^3 - 4 pi r^3
double vol = 4.0*pi*R*R*R/3.0 - 4.0*pi*r*r*r/3.0;

//Addition:

int a = 2+4/2;          // equal to: 2+(4/2)         result: 4
int b = (3+3)/2;        // equal to: (3+3)/2         result: 3

//With Multiplication

int c = 3+4/2*6;        // equal to: 3+((4/2)*6)     result: 15
int d = 3*(3+6)/9;      // equal to: (3*(3+6))/9     result: 3

//Division and Modulo

int g = 3-3%1;          // equal to: 3 % 1 = 0  3 - 0 = 3
int h = 3-(3%1);        // equal to: 3 % 1 = 0  3 - 0 = 3
int i = 3-3/1%3;        // equal to: 3 / 1 = 3  3 % 3 = 0  3 - 0 = 3
int l = 3-(3/1)%3;      // equal to: 3 / 1 = 3  3 % 3 = 0  3 - 0 = 3
int m = 3-(3/(1%3));    // equal to: 1 % 3 = 1  3 / 1 = 3  3 - 3 = 0
```

Logische UND- und ODER-Operatoren

Diese Operatoren haben die übliche Priorität in C ++: AND vor OR.

```
// You can drive with a foreign license for up to 60 days
bool can_drive = has_domestic_license || has_foreign_license && num_days <= 60;
```

Dieser Code entspricht dem folgenden:

```
// You can drive with a foreign license for up to 60 days
bool can_drive = has_domestic_license || (has_foreign_license && num_days <= 60);
```

Durch das Hinzufügen der Klammer wird das Verhalten nicht geändert, es wird jedoch einfacher lesbar. Durch das Hinzufügen dieser Klammern besteht keine Verwirrung hinsichtlich der Absicht des Verfassers.

Logisch && und || Betreiber: Kurzschluss

&& hat Vorrang vor ||, das bedeutet, dass Klammern gesetzt werden, um auszuwerten, was zusammen ausgewertet würde.

c ++ verwendet die Kurzschlussbewertung in && und || keine unnötigen Hinrichtungen durchführen.
Wenn die linke Seite von || Gibt true zurück, die rechte Seite muss nicht mehr ausgewertet werden.

```
#include <iostream>
#include <string>

using namespace std;

bool True(string id){
    cout << "True" << id << endl;
    return true;
}

bool False(string id){
    cout << "False" << id << endl;
    return false;
}


int main(){
    bool result;
    //let's evaluate 3 booleans with || and && to illustrate operator precedence
    //precedence does not mean that && will be evaluated first but rather where
    //parentheses would be added
    //example 1
    result =
        False("A") || False("B") && False("C");
                // eq. False("A") || (False("B") && False("C"))
    //FalseA
    //FalseB
    //"Short-circuit evaluation skip of C"
    //A is false so we have to evaluate the right of ||,
    //B being false we do not have to evaluate C to know that the result is false



    result =
        True("A") || False("B") && False("C");
                // eq. True("A") || (False("B") && False("C"))
    cout << result << " :====================" << endl;
    //TrueA
    //"Short-circuit evaluation skip of B"
    //"Short-circuit evaluation skip of C"
    //A is true so we do not have to evaluate
    //        the right of || to know that the result is true
    //If || had precedence over && the equivalent evaluation would be:
    // (True("A") || False("B")) && False("C")
    //What would print
    //TrueA
    //"Short-circuit evaluation skip of B"
    //FalseC
    //Because the parentheses are placed differently
    //the parts that get evaluated are differently
    //which makes that the end result in this case would be False because C is false
}
```

Unäre Operatoren

Unäre Operatoren wirken auf das Objekt ein, auf das sie aufgerufen werden und haben eine hohe Priorität. (Siehe Anmerkungen)

Wenn Postfix verwendet wird, wird die Aktion erst ausgeführt, nachdem die gesamte Operation ausgewertet wurde, was zu einigen interessanten Arithmetiken führt:

```
int a = 1;
++a;             // result: 2
a--;             // result: 1
int minusa=-a;   // result: -1

bool b = true;
!b; // result: true

a=4;
int c = a++/2;       // equal to: (a==4) 4 / 2   result: 2 ('a' incremented postfix)
cout << a << endl;   // prints 5!
int d = ++a/2;       // equal to: (a+1) == 6 / 2 result: 3

int arr[4] =  {1,2,3,4};

int *ptr1 = &arr[0];    // points to arr[0] which is 1
int *ptr2 = ptr1++;     // ptr2 points to arr[0] which is still 1; ptr1 incremented
std::cout << *ptr1++ << std::endl;  // prints  2

int e = arr[0]++;       // receives the value of arr[0] before it is incremented
std::cout << e << std::endl;      // prints 1
std::cout << *ptr2 << std::endl;  // prints arr[0] which is now 2
```

Operator Vorrang online lesen: https://riptutorial.com/de/cplusplus/topic/3895/operator-vorrang

Kapitel 80: Optimierung

Einführung

Beim Compilieren ändert der Compiler das Programm häufig, um die Leistung zu erhöhen. Dies wird durch die Ist-Regel erlaubt, die alle Transformationen erlaubt, die das beobachtbare Verhalten nicht ändern.

Examples

Inline-Erweiterung / Inlining

Inline-Erweiterung (auch Inlining genannt) ist die Compiler-Optimierung, die einen Aufruf einer Funktion durch den Rumpf dieser Funktion ersetzt. Dies spart den Funktionsaufruf-Overhead, jedoch auf Kosten des Platzbedarfs, da die Funktion möglicherweise mehrmals dupliziert wird.

```
// source:

int process(int value)
{
    return 2 * value;
}

int foo(int a)
{
    return process(a);
}

// program, after inlining:

int foo(int a)
{
    return 2 * a; // the body of process() is copied into foo()
}
```

Das Inlining wird am häufigsten für kleine Funktionen ausgeführt, bei denen der Funktionsaufruf für die Funktionsaufrufe im Vergleich zur Größe des Funktionskörpers erheblich ist.

Leere Basisoptimierung

Die Größe eines Objekt- oder Member-Subobjekts muss mindestens 1 sein, selbst wenn der Typ ein leerer `class` (d. `struct` Eine `class` oder `struct` , die keine nicht statischen Datenelemente enthält), um dies gewährleisten zu können Die Adressen verschiedener Objekte desselben Typs sind immer unterschiedlich.

Basisklassen `class` Unterobjekte sind jedoch nicht so eingeschränkt und können vollständig aus dem Objektlayout heraus optimiert werden:

```
#include <cassert>
```

```
struct Base {}; // empty class

struct Derived1 : Base {
    int i;
};

int main() {
    // the size of any object of empty class type is at least 1
    assert(sizeof(Base) == 1);

    // empty base optimization applies
    assert(sizeof(Derived1) == sizeof(int));
}
```

Eine leere Basisoptimierung wird im Allgemeinen von zuweisungspflichtigen Standardbibliotheksklassen (`std::vector` , `std::function` , `std::shared_ptr` usw.) verwendet, um zu vermeiden, dass zusätzlicher Speicher für das Zuordnungsmitglied belegt wird, wenn der Zuweiser zustandslos ist. Dies wird erreicht, indem eines der erforderlichen Datenelemente (z. B. `begin` , `end` oder `capacity` für den `vector`) gespeichert wird.

Referenz: cppreference

Optimierung online lesen: https://riptutorial.com/de/cplusplus/topic/9767/optimierung

Kapitel 81: Optimierung in C ++

Examples

Leere Basisklassenoptimierung

Ein Objekt darf nicht weniger als 1 Byte belegen, da dann die Mitglieder eines Arrays dieses Typs dieselbe Adresse hätten. Also gilt `sizeof(T)>=1` immer. Es ist auch wahr , dass eine abgeleitete Klasse nicht als *eine* ihrer Basisklassen kleiner sein kann. Wenn die Basisklasse jedoch leer ist, wird ihre Größe nicht notwendigerweise zur abgeleiteten Klasse hinzugefügt:

```
class Base {};

class Derived : public Base
{
public:
    int i;
};
```

In diesem Fall ist es nicht erforderlich, ein Byte für `Base` in `Derived` zuzuordnen, um pro Objekt und Objekt eine eindeutige Adresse zu erhalten. Wenn eine leere Basisklassenoptimierung durchgeführt wird (und keine Auffüllung erforderlich ist), dann `sizeof(Derived) == sizeof(int)` , `sizeof(Derived) == sizeof(int)` , es wird keine zusätzliche Zuordnung für die leere Basis vorgenommen. Dies ist auch mit mehreren Basisklassen möglich (in C ++ können nicht mehrere Basen denselben Typ haben, daher treten keine Probleme auf).

Beachten Sie, dass dies nur möglich ist, wenn der erste Member von `Derived` sich von den Basisklassen in seinem Typ unterscheidet. Dies schließt jegliche direkte oder indirekte Basis ein. Wenn es sich um denselben Typ handelt wie eine der Basen (oder es gibt eine gemeinsame Basis), muss mindestens ein Byte zugeordnet werden, um sicherzustellen, dass keine zwei verschiedenen Objekte desselben Typs dieselbe Adresse haben.

Einführung in die Leistung

C und C ++ sind allgemein als Hochleistungssprachen bekannt - hauptsächlich aufgrund der umfangreichen Anpassungen von Code, die es einem Benutzer ermöglichen, die Leistung durch die Wahl der Struktur festzulegen.

Bei der Optimierung ist es wichtig, den relevanten Code zu vergleichen und zu verstehen, wie der Code verwendet wird.

Häufige Optimierungsfehler sind:

- **Vorzeitige Optimierung:** Komplexer Code kann nach der Optimierung *schlechter* ablaufen, was Zeit und Mühe kostet. Die erste Priorität sollte das Schreiben von *korrektem* und *wartungsfähigem* Code anstelle von optimiertem Code sein.
- **Optimierung für den falschen Anwendungsfall:** Das Hinzufügen von Overhead für die 1%

ist möglicherweise nicht die Verlangsamung für die anderen 99% wert.
- **Mikrooptimierung:** Compiler machen dies sehr effizient, und Mikrooptimierung kann sogar die Fähigkeit des Compilers beeinträchtigen, den Code weiter zu optimieren

Typische Optimierungsziele sind:

- Weniger arbeiten
- Effizientere Algorithmen / Strukturen verwenden
- Hardware besser nutzen

Optimierter Code kann negative Nebenwirkungen haben, einschließlich:

- Höhere Speicherauslastung
- Komplexer Code - schwer zu lesen oder zu warten
- Kompromissloses API- und Code-Design

Optimierung durch weniger Code ausführen

Der einfachste Ansatz zur Optimierung besteht darin, weniger Code auszuführen. Dieser Ansatz führt normalerweise zu einer festen Beschleunigung, ohne die zeitliche Komplexität des Codes zu ändern.

Auch wenn dieser Ansatz eine deutliche Beschleunigung bewirkt, führt dies nur zu spürbaren Verbesserungen, wenn der Code viel aufgerufen wird.

Nutzlosen Code entfernen

```
void func(const A *a); // Some random function

// useless memory allocation + deallocation for the instance
auto a1 = std::make_unique<A>();
func(a1.get());

// making use of a stack object prevents
auto a2 = A{};
func(&a2);
```

C ++ 14

Ab C ++ 14 dürfen Compiler diesen Code optimieren, um die Zuordnung und die Zuordnung der Zuweisung zu entfernen.

Code nur einmal machen

```
std::map<std::string, std::unique_ptr<A>> lookup;
// Slow insertion/lookup
// Within this function, we will traverse twice through the map lookup an element
// and even a thirth time when it wasn't in
const A *lazyLookupSlow(const std::string &key) {
    if (lookup.find(key) != lookup.cend())
```

```
        lookup.emplace_back(key, std::make_unique<A>());
    return lookup[key].get();
}

// Within this function, we will have the same noticeable effect as the slow variant while
going at double speed as we only traverse once through the code
const A *lazyLookupSlow(const std::string &key) {
    auto &value = lookup[key];
    if (!value)
        value = std::make_unique<A>();
    return value.get();
}
```

Ein ähnlicher Ansatz zu dieser Optimierung kann verwendet werden, um eine stabile Version von `unique` zu implementieren

```
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::set<std::string> checkUnique;
    for (const auto &s : v) {
        // As insert returns if the insertion was successful, we can deduce if the element was
already in or not
        // This prevents an insertion, which will traverse through the map for every unique
element
        // As a result we can almost gain 50% if v would not contain any duplicates
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

Verhindern Sie unnötige Neuzuordnungen und Kopieren / Verschieben

Im vorherigen Beispiel haben wir bereits Lookups in std :: set verhindert, der `std::vector` enthält jedoch noch einen wachsenden Algorithmus, in dem er seinen Speicher neu zuordnen muss. Dies kann verhindert werden, indem zunächst die richtige Größe reserviert wird.

```
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    // By reserving 'result', we can ensure that no copying or moving will be done in the
vector
    // as it will have capacity for the maximum number of elements we will be inserting
    // If we make the assumption that no allocation occurs for size zero
    // and allocating a large block of memory takes the same time as a small block of memory
    // this will never slow down the program
    // Side note: Compilers can even predict this and remove the checks the growing from the
generated code
    result.reserve(v.size());
    std::set<std::string> checkUnique;
    for (const auto &s : v) {
        // See example above
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
```

```
    return result;
}
```

Effiziente Behälter verwenden

Durch die Optimierung der richtigen Datenstrukturen zum richtigen Zeitpunkt kann sich die zeitliche Komplexität des Codes ändern.

```
// This variant of stableUnique contains a complexity of N log(N)
// N > number of elements in v
// log(N) > insert complexity of std::set
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::set<std::string> checkUnique;
    for (const auto &s : v) {
        // See Optimizing by executing less code
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

Durch die Verwendung eines Containers, der eine andere Implementierung zum Speichern seiner Elemente verwendet (Hash-Container statt Baum), können wir unsere Implementierung in Komplexität N umwandeln. Als Nebeneffekt rufen wir den Vergleichsoperator für std :: string less auf muss nur aufgerufen werden, wenn der eingefügte String im selben Bucket enden soll.

```
// This variant of stableUnique contains a complexity of N
// N > number of elements in v
// 1 > insert complexity of std::unordered_set
std::vector<std::string> stableUnique(const std::vector<std::string> &v) {
    std::vector<std::string> result;
    std::unordered_set<std::string> checkUnique;
    for (const auto &s : v) {
        // See Optimizing by executing less code
        if (checkUnique.insert(s).second)
            result.push_back(s);
    }
    return result;
}
```

Kleinobjekt-Optimierung

Die Optimierung von kleinen Objekten ist eine Technik, die in Datenstrukturen niedriger Ebene verwendet wird, z. B. `std::string` (manchmal auch als kurze / kleine String-Optimierung bezeichnet). Es ist dazu gedacht, Stapelspeicher als Puffer zu verwenden, anstelle von zugewiesenem Speicher, falls der Inhalt klein genug ist, um in den reservierten Speicherplatz zu passen.

Durch das Hinzufügen von zusätzlichem Speicheraufwand und zusätzlichen Berechnungen wird versucht, eine teure Heapzuweisung zu verhindern. Die Vorteile dieser Technik hängen von der Verwendung ab und können bei falscher Verwendung sogar die Leistung beeinträchtigen.

Beispiel

Eine sehr naive Art, eine Zeichenfolge mit dieser Optimierung zu implementieren, wäre folgende:

```
#include <cstring>

class string final
{
    constexpr static auto SMALL_BUFFER_SIZE = 16;

    bool _isAllocated{false};                       ///< Remember if we allocated memory
    char *_buffer{nullptr};                         ///< Pointer to the buffer we are using
    char _smallBuffer[SMALL_BUFFER_SIZE]= {'\0'};   ///< Stack space used for SMALL OBJECT
OPTIMIZATION

public:
    ~string()
    {
        if (_isAllocated)
            delete [] _buffer;
    }

    explicit string(const char *cStyleString)
    {
        auto stringSize = std::strlen(cStyleString);
        _isAllocated = (stringSize > SMALL_BUFFER_SIZE);
        if (_isAllocated)
            _buffer = new char[stringSize];
        else
            _buffer = &_smallBuffer[0];
        std::strcpy(_buffer, &cStyleString[0]);
    }

    string(string &&rhs)
       : _isAllocated(rhs._isAllocated)
       , _buffer(rhs._buffer)
       , _smallBuffer(rhs._smallBuffer) //< Not needed if allocated
    {
        if (_isAllocated)
        {
           // Prevent double deletion of the memory
           rhs._buffer = nullptr;
        }
        else
        {
            // Copy over data
            std::strcpy(_smallBuffer, rhs._smallBuffer);
            _buffer = &_smallBuffer[0];
        }
    }
    // Other methods, including other constructors, copy constructor,
    // assignment operators have been omitted for readability
};
```

Wie Sie im obigen Code sehen können, wurde etwas mehr Komplexität hinzugefügt, um einige `new` und `delete` zu verhindern. Darüber hinaus verfügt die Klasse über einen größeren Speicherbedarf, der möglicherweise nur in einigen Fällen verwendet wird.

Oft wird versucht , den Wert zu codieren bool `_isAllocated` innerhalb des Zeigers `_buffer` mit Bitmanipulation die Größe einer einzelnen Instanz (Intel - 64 - Bit: kann Größe reduzieren um 8 Byte) zu reduzieren. Eine Optimierung, die nur möglich ist, wenn die Ausrichtungsregeln der Plattform bekannt sind.

Wann verwenden?

Da diese Optimierung die Komplexität erhöht, wird nicht empfohlen, diese Optimierung für jede einzelne Klasse zu verwenden. Sie wird häufig in häufig verwendeten Datenstrukturen niedriger Ebene angetroffen. In gängigen C ++ 11- `standard library` finden Sie Verwendungen in `std::basic_string<>` und `std::function<>` .

Da diese Optimierung nur Speicherzuordnungen verhindert, wenn die gespeicherten Daten kleiner als der Puffer sind, bietet dies nur Vorteile, wenn die Klasse häufig mit kleinen Daten verwendet wird.

Ein letzter Nachteil dieser Optimierung besteht darin, dass beim Verschieben des Puffers ein zusätzlicher Aufwand erforderlich ist, was den Verschiebevorgang teurer macht, als wenn der Puffer nicht verwendet würde. Dies gilt insbesondere, wenn der Puffer einen Nicht-POD-Typ enthält.

Optimierung in C ++ online lesen: https://riptutorial.com/de/cplusplus/topic/4474/optimierung-in-c-plusplus

Kapitel 82: Parallele Vergleiche von klassischen C ++ - Beispielen, die über C ++ vs C ++ 11 vs C ++ 14 vs C ++ 17 gelöst wurden

Examples

Durchlaufen eines Containers

In C ++ kann das Durchlaufen eines Sequenzcontainers `c` mithilfe der folgenden Indizes durchgeführt werden:

```
for(size_t i = 0; i < c.size(); ++i) c[i] = 0;
```

Während diese Schriften einfach sind, unterliegen sie häufigen semantischen Fehlern wie einem falschen Vergleichsoperator oder einer falschen Indizierungsvariablen:

```
for(size_t i = 0; i <= c.size(); ++j) c[i] = 0;
                     ^~~~~~~~~~~~~~^
```

Eine Schleife kann auch für alle Container mit Iteratoren mit ähnlichen Nachteilen erreicht werden:

```
for(iterator it = c.begin(); it != c.end(); ++it) (*it) = 0;
```

In C ++ 11 wurden bereichsbasierte für Schleifen und `auto` Schlüsselwörter eingeführt, wodurch der Code zu Folgendem werden konnte:

```
for(auto& x : c) x = 0;
```

Hier sind die einzigen Parameter der Container `c` und eine Variable `x` , die den aktuellen Wert enthält. Dies verhindert die zuvor angesprochenen Semantikfehler.

Gemäß dem C ++ 11-Standard entspricht die zugrunde liegende Implementierung:

```
for(auto begin = c.begin(), end = c.end(); begin != end; ++begin)
{
    // ...
}
```

In einer solchen Implementierung ist der Ausdruck `auto begin = c.begin(), end = c.end();` Kräfte `begin` und `end` als vom selben Typ, während das `end` niemals inkrementiert oder dereferenziert wird. Die Range-basierte for-Schleife funktioniert also nur für Container, die von einem Paar-Iterator / Iterator definiert werden. Der C ++ 17-Standard mildert diese Einschränkung, indem die

Implementierung folgendermaßen geändert wird:

```
auto begin = c.begin();
auto end = c.end();
for(; begin != end; ++begin)
{
    // ...
}
```

`begin` und `end` dürfen hier von unterschiedlichen Typen sein, solange sie auf Ungleichheit verglichen werden können. Dies ermöglicht das Durchlaufen mehrerer Container, z. B. eines Containers, der von einem Paar-Iterator / Sentinel definiert wird.

Parallele Vergleiche von klassischen C ++ - Beispielen, die über C ++ vs C ++ 11 vs C ++ 14 vs C ++ 17 gelöst wurden online lesen: https://riptutorial.com/de/cplusplus/topic/7134/parallele-vergleiche-von-klassischen-c-plusplus---beispielen--die-uber-c-plusplus-vs-c-plusplus-11-vs-c-plusplus-14-vs-c-plusplus-17-gelost-wurden

Kapitel 83: Parallelität mit OpenMP

Einführung

Dieses Thema behandelt die Grundlagen der Parallelität in C ++ mit OpenMP. OpenMP ist im OpenMP-Tag ausführlicher dokumentiert.

Parallelität oder Parallelität impliziert die gleichzeitige Ausführung von Code.

Bemerkungen

OpenMP erfordert keine speziellen Header oder Bibliotheken, da es sich um eine integrierte Compiler-Funktion handelt. Wenn Sie jedoch OpenMP-API-Funktionen wie `omp_get_thread_num()` , müssen Sie `omp.h` und seine Bibliothek `omp.h` .

OpenMP- `pragma` Anweisungen werden ignoriert, wenn die OpenMP-Option während der Kompilierung nicht aktiviert ist. Sie können auf die Compiler-Option im Handbuch Ihres Compilers verweisen.

- GCC verwendet `-fopenmp`
- Clang verwendet `-fopenmp`
- MSVC verwendet `/openmp`

Examples

OpenMP: Parallele Abschnitte

Dieses Beispiel veranschaulicht die Grundlagen für die parallele Ausführung von Codeabschnitten.

Da OpenMP eine integrierte Compiler-Funktion ist, kann es auf jedem unterstützten Compiler ohne Bibliotheken verwendet werden. Sie können `omp.h` wenn Sie die openMP-API-Funktionen verwenden möchten.

Beispielcode

```
std::cout << "begin ";
//    This pragma statement hints the compiler that the
//    contents within the { } are to be executed in as
//    parallel sections using openMP, the compiler will
//    generate this chunk of code for parallel execution
#pragma omp parallel sections
{
    //    This pragma statement hints the compiler that
    //    this is a section that can be executed in parallel
    //    with other section, a single section will be executed
    //    by a single thread.
    //    Note that it is "section" as opposed to "sections" above
```

```
    #pragma omp section
    {
        std::cout << "hello " << std::endl;
        /** Do something **/
    }
    #pragma omp section
    {
        std::cout << "world " << std::endl;
        /** Do something **/
    }
}
//    This line will not be executed until all the
//    sections defined above terminates
std::cout << "end" << std::endl;
```

Ausgänge

Dieses Beispiel liefert 2 mögliche Ausgänge und ist abhängig von Betriebssystem und Hardware. Die Ausgabe zeigt auch ein **Race-Condition-** Problem, das bei einer solchen Implementierung auftreten würde.

AUSGABE A	AUSGABE B
begin hallo welt ende	Welt beginnen Hallo Ende

OpenMP: Parallele Abschnitte

Dieses Beispiel zeigt, wie Codeabschnitte parallel ausgeführt werden

```
std::cout << "begin ";
//    Start of parallel sections
#pragma omp parallel sections
{
    //    Execute these sections in parallel
    #pragma omp section
    {
        ... do something ...
        std::cout << "hello ";
    }
    #pragma omp section
    {
        ... do something ...
        std::cout << "world ";
    }
    #pragma omp section
    {
        ... do something ...
        std::cout << "forever ";
    }
}
//    end of parallel sections
std::cout << "end";
```

Ausgabe

- beginnen hallo welt für immer ende
- Welt beginnen Hallo für immer Ende
- fang hallo für immer das weltende an
- für immer hallo Weltende beginnen

Da die Ausführungsreihenfolge nicht garantiert werden kann, können Sie eine der obigen Ausgaben beobachten.

OpenMP: Parallel für Schleife

Dieses Beispiel zeigt, wie eine Schleife in gleiche Teile aufgeteilt und parallel ausgeführt wird.

```
//    Splits element vector into element.size() / Thread Qty
//    and allocate that range for each thread.
#pragma omp parallel for
for    (size_t i = 0; i < element.size(); ++i)
    element[i] = ...

//    Example Allocation (100 element per thread)
//    Thread 1 : 0 ~ 99
//    Thread 2 : 100 ~ 199
//    Thread 2 : 200 ~ 299
//    ...

//    Continue process
//    Only when all threads completed their allocated
//    loop job
...
```

* Bitte achten Sie besonders darauf, die Größe des für Schleifen parallel verwendeten Vektors nicht zu ändern, da **zugeordnete Bereichsindizes nicht automatisch aktualisiert werden** .

OpenMP: Parallele Erfassung / Reduzierung

Dieses Beispiel veranschaulicht ein Konzept zum Durchführen einer Reduktion oder Erfassung mit `std::vector` und OpenMP.

Angenommen, wir haben ein Szenario, in dem wir möchten, dass mehrere Threads uns helfen, ein paar Sachen zu generieren. `int` wird hier der Einfachheit halber verwendet und kann durch andere Datentypen ersetzt werden.

Dies ist besonders nützlich, wenn Sie die Ergebnisse von Slaves zusammenführen müssen, um Segmentfehler oder Verstöße gegen den Speicherzugriff zu vermeiden und keine Bibliotheken oder benutzerdefinierte Sync-Container-Bibliotheken verwenden möchten.

```
//    The Master vector
//    We want a vector of results gathered from slave threads
std::vector<int> Master;

//    Hint the compiler to parallelize this { } of code
//    with all available threads (usually the same as logical processor qty)
#pragma omp parallel
{
```

```
    //    In this area, you can write any code you want for each
    //    slave thread, in this case a vector to hold each of their results
    //    We don't have to worry about how many threads were spawn or if we need
    //    to repeat this declaration or not.
    std::vector<int> Slave;

    //    Tell the compiler to use all threads allocated for this parallel region
    //    to perform this loop in parts. Actual load appx = 1000000 / Thread Qty
    //    The nowait keyword tells the compiler that the slave threads don't
    //    have to wait for all other slaves to finish this for loop job
    #pragma omp for nowait
    for (size_t i = 0; i < 1000000; ++i
    {
        /* Do something */
        ....
        Slave.push_back(...);
    }

    //    Slaves that finished their part of the job
    //    will perform this thread by thread one at a time
    //    critical section ensures that only 0 or 1 thread performs
    //    the { } at any time
    #pragma omp critical
    {
        //    Merge slave into master
        //    use move iterators instead, avoid copy unless
        //    you want to use it for something else after this section
        Master.insert(Master.end(),
                      std::make_move_iterator(Slave.begin()),
                      std::make_move_iterator(Slave.end()));
    }
}

//    Have fun with Master vector
...
```

Parallelität mit OpenMP online lesen: https://riptutorial.com/de/cplusplus/topic/8222/parallelitat-mit-openmp

Kapitel 84: Parameterpakete

Examples

Eine Vorlage mit einem Parameterpaket

```
template<class ... Types> struct Tuple {};
```

Ein Parameterpaket ist ein Vorlagenparameter, der null oder mehr Vorlagenargumente akzeptiert. Wenn eine Vorlage mindestens ein Parameterpaket enthält, handelt es sich um eine *variadische Vorlage* .

Erweiterung eines Parameterpakets

Das Muster `parameter_pack ...` wird zu einer Liste von durch Kommas getrennten Ersetzungen von `parameter_pack` mit jedem seiner Parameter erweitert

```
template<class T> // Base of recursion
void variadic_printer(T last_argument) {
    std::cout << last_argument;
}

template<class T, class ...Args>
void variadic_printer(T first_argument, Args... other_arguments) {
  std::cout << first_argument << "\n";
  variadic_printer(other_arguments...); // Parameter pack expansion
}
```

Der oben genannte Code wurde mit `variadic_printer(1, 2, 3, "hello");` aufgerufen `variadic_printer(1, 2, 3, "hello");` druckt

```
1
2
3
hello
```

Parameterpakete online lesen: https://riptutorial.com/de/cplusplus/topic/7668/parameterpakete

Kapitel 85: Perfekte Weiterleitung

Bemerkungen

Eine einwandfreie Weiterleitung erfordert *Weiterleitungsreferenzen* , um die Ref-Qualifier der Argumente zu erhalten. Solche Verweise erscheinen nur in einem *abgeleiteten Kontext* . Das ist:

```
template<class T>
void f(T&& x) // x is a forwarding reference, because T is deduced from a call to f()
{
    g(std::forward<T>(x)); // g() will receive an lvalue or an rvalue, depending on x
}
```

Folgendes beinhaltet keine perfekte Weiterleitung, da `T` nicht vom Konstruktoraufruf abgeleitet wird:

```
template<class T>
struct a
{
    a(T&& x); // x is a rvalue reference, not a forwarding reference
};
```

C ++ 17

In C ++ 17 können Klassenvorlagenargumente abgezogen werden. Der Konstruktor von "a" im obigen Beispiel wird Benutzer einer Weiterleitungsreferenz

```
a example1(1);
  // same as a<int> example1(1);

int x = 1;
a example2(x);
  // same as a<int&> example2(x);
```

Examples

Werksfunktionen

Angenommen, wir möchten eine Factory-Funktion schreiben, die eine beliebige Liste von Argumenten akzeptiert und diese Argumente unverändert an eine andere Funktion übergibt. Ein Beispiel für eine solche Funktion ist `make_unique` , mit der eine neue Instanz von `T` sicher erstellt und ein `unique_ptr<T>` , das die Instanz besitzt.

Die Sprachregeln für Variadic-Templates und -Referenzreferenzen ermöglichen das Schreiben einer solchen Funktion.

```
template<class T, class... A>
unique_ptr<T> make_unique(A&&... args)
```

```
{
    return unique_ptr<T>(new T(std::forward<A>(args)...));
}
```

Die Verwendung von Ellipsen `...` gibt ein Parameterpaket an, das eine beliebige Anzahl von Typen darstellt. Der Compiler erweitert dieses Parameterpaket auf die richtige Anzahl von Argumenten an der Aufrufstelle. Diese Argumente werden dann mit `std::forward` Konstruktor von `T` . Diese Funktion ist erforderlich, um die Ref-Qualifier der Argumente zu erhalten.

```
struct foo
{
    foo() {}
    foo(const foo&) {}                    // copy constructor
    foo(foo&&) {}                         // copy constructor
    foo(int, int, int) {}
};

foo f;
auto p1 = make_unique<foo>(f);            // calls foo::foo(const foo&)
auto p2 = make_unique<foo>(std::move(f)); // calls foo::foo(foo&&)
auto p3 = make_unique<foo>(1, 2, 3);
```

Perfekte Weiterleitung online lesen: https://riptutorial.com/de/cplusplus/topic/1750/perfekte-weiterleitung

Kapitel 86: Pimpl-Idiom

Bemerkungen

Das **Pimpl-Idiom** (**P** ointer to **Implementation** , manchmal auch als *undurchsichtiger Zeiger* oder *Cheshire-Cat-Verfahren bezeichnet*) reduziert die Kompilierungszeiten einer Klasse, indem alle privaten Datenelemente in eine in der CPP-Datei definierte Struktur verschoben werden.

Die Klasse besitzt einen Zeiger auf die Implementierung. Auf diese Weise kann es vorwärts erklärt werden, so dass die Header - Datei nicht benötigt `#include` - Klassen , die in der privaten Membervariablen verwendet werden.

Bei Verwendung des Pimpl-Idioms müssen für das Ändern eines privaten Datenelements keine davon abhängigen Klassen neu kompiliert werden.

Examples

Grundlegendes Pimpl-Idiom

C ++ 11

In der Headerdatei:

```
// widget.h

#include <memory>  // std::unique_ptr
#include <experimental/propagate_const>

class Widget
{
    public:
        Widget();
        ~Widget();
        void DoSomething();

    private:
        // the pImpl idiom is named after the typical variable name used
        // ie, pImpl:
        struct Impl;                    // forward declaration
        std::experimental::propagate_const<std::unique_ptr< Impl >> pImpl;  // ptr to actual
implementation
};
```

In der Implementierungsdatei:

```
// widget.cpp

#include "widget.h"
#include "reallycomplextype.h" // no need to include this header inside widget.h
```

```
struct Widget::Impl
{
    // the attributes needed from Widget go here
    ReallyComplexType rct;
};

Widget::Widget() :
    pImpl(std::make_unique<Impl>())
{}

Widget::~Widget() = default;

void Widget::DoSomething()
{
    // do the stuff here with pImpl
}
```

Das `pImpl` enthält den `Widget` Status (oder einige / die meisten davon). Anstatt die `Widget` in der Headerdatei verfügbar zu machen, kann sie nur innerhalb der Implementierung verfügbar gemacht werden.

`pImpl` steht für "Zeiger auf Implementierung". Die "echte" Implementierung von `Widget` befindet sich im `pImpl` .

Gefahr: Beachten Sie, dass `~Widget()` an einer Stelle in einer Datei implementiert werden muss, an der das `Impl` vollständig sichtbar ist, damit dies mit `unique_ptr Impl` . Sie können `=default` es gibt, aber wenn `=default` wo `Impl` nicht definiert ist, wird das Programm leicht schlecht ausgebildet ist , keine Diagnose erforderlich.

Pimpl-Idiom online lesen: https://riptutorial.com/de/cplusplus/topic/2143/pimpl-idiom

Kapitel 87: Polymorphismus

Examples

Definieren Sie polymorphe Klassen

Das typische Beispiel ist eine abstrakte Formklasse, die dann in Quadrate, Kreise und andere konkrete Formen abgeleitet werden kann.

Die übergeordnete Klasse:

Beginnen wir mit der polymorphen Klasse:

```
class Shape {
public:
    virtual ~Shape() = default;
    virtual double get_surface() const = 0;
    virtual void describe_object() const { std::cout << "this is a shape" << std::endl; }

    double get_doubled_surface() const { return 2 * get_surface(); }
};
```

Wie kann man diese Definition lesen?

- Sie können polymorphes Verhalten durch eingeführte Elementfunktionen mit dem Schlüsselwort `virtual` . Hier `get_surface()` und `describe_object()` wird offensichtlich für ein Quadrat unterschiedlich implementiert werden , als für einen Kreis. Wenn die Funktion für ein Objekt aufgerufen wird, wird zur Laufzeit eine Funktion bestimmt, die der realen Klasse des Objekts entspricht.
- Es macht keinen Sinn, `get_surface()` für eine abstrakte Form zu definieren. Deshalb folgt auf die Funktion `= 0` . Dies bedeutet, dass die Funktion eine *rein virtuelle Funktion ist* .
- Eine polymorphe Klasse sollte immer einen virtuellen Destruktor definieren.
- Sie können nicht virtuelle Elementfunktionen definieren. Wenn diese Funktion für ein Objekt aufgerufen wird, wird die Funktion in Abhängigkeit von der zur Kompilierzeit verwendeten Klasse ausgewählt. Hier ist `get_double_surface()` so definiert.
- Eine Klasse, die mindestens eine rein virtuelle Funktion enthält, ist eine abstrakte Klasse. Abstrakte Klassen können nicht instanziiert werden. Sie können nur Zeiger oder Referenzen eines abstrakten Klassentyps haben.

Abgeleitete Klassen

Sobald eine polymorphe Basisklasse definiert ist, können Sie sie ableiten. Zum Beispiel:

```
class Square : public Shape {
    Point top_left;
```

```
    double side_length;
public:
    Square (const Point& top_left, double side)
       : top_left(top_left), side_length(side_length) {}

    double get_surface() override { return side_length * side_length; }
    void describe_object() override {
        std::cout << "this is a square starting at " << top_left.x << ", " << top_left.y
                  << " with a length of " << side_length << std::endl;
    }
};
```

Einige Erklärungen:

- Sie können alle virtuellen Funktionen der übergeordneten Klasse definieren oder überschreiben. Die Tatsache, dass eine Funktion in der übergeordneten Klasse virtuell war, macht sie in der abgeleiteten Klasse virtuell. Dem Compiler muss das Schlüsselwort `virtual` einmal `virtual` werden. Aber es wird empfohlen , das Keyword hinzufügen `override` am Ende der Funktionsdeklaration, um geringfügige Fehler unbemerkt Variationen in der Funktionssignatur zu verhindern.
- Wenn alle reinen virtuellen Funktionen der übergeordneten Klasse definiert sind, können Sie Objekte für diese Klasse instanziieren, andernfalls wird sie auch zu einer abstrakten Klasse.
- Sie sind nicht verpflichtet, alle virtuellen Funktionen zu überschreiben. Sie können die Version des übergeordneten Elements beibehalten, wenn es Ihren Bedürfnissen entspricht.

Beispiel für eine Instantiierung

```
int main() {

    Square square(Point(10.0, 0.0), 6); // we know it's a square, the compiler also
    square.describe_object();
    std::cout << "Surface: " << square.get_surface() << std::endl;

    Circle circle(Point(0.0, 0.0), 5);

    Shape *ps = nullptr;  // we don't know yet the real type of the object
    ps = &circle;         // it's a circle, but it could as well be a square
    ps->describe_object();
    std::cout << "Surface: " << ps->get_surface() << std::endl;
}
```

Sicheres Downcasting

Angenommen, Sie haben einen Zeiger auf ein Objekt einer polymorphen Klasse:

```
Shape *ps;                        // see example on defining a polymorphic class
ps =  get_a_new_random_shape();   // if you don't have such a function yet, you
                                  // could just write ps = new Square(0.0,0.0, 5);
```

Ein Downcast wäre, von einer allgemeinen polymorphen `Shape` auf eine abgeleitete und spezifischere Form wie `Square` oder `Circle` .

Warum niedergeschlagen?

In den meisten Fällen müssen Sie nicht wissen, um welchen realen Objekttyp es sich handelt, da Sie mit den virtuellen Funktionen Ihr Objekt unabhängig von seinem Objekttyp bearbeiten können:

```
std::cout << "Surface: " << ps->get_surface() << std::endl;
```

Wenn Sie keinen Downcast benötigen, wäre Ihr Design perfekt.

Möglicherweise müssen Sie jedoch manchmal einen Downcast durchführen. Ein typisches Beispiel ist, wenn Sie eine nicht virtuelle Funktion aufrufen möchten, die nur für die untergeordnete Klasse vorhanden ist.

Betrachten Sie zum Beispiel Kreise. Nur Kreise haben einen Durchmesser. Die Klasse wäre also definiert als:

```
class Circle: public Shape { // for Shape, see example on defining a polymorphic class
    Point center;
    double radius;
public:
    Circle (const Point& center, double radius)
       : center(center), radius(radius) {}

    double get_surface() const override { return r * r * M_PI; }

    // this is only for circles. Makes no sense for other shapes
    double get_diameter() const { return 2 * r; }
};
```

Die Member-Funktion `get_diameter()` existiert nur für Kreise. Es wurde nicht für ein `Shape` Objekt definiert:

```
Shape* ps = get_any_shape();
ps->get_diameter(); // OUCH !!! Compilation error
```

Wie niederzuschlagen?

Wenn Sie sicher wissen, dass `ps` auf einen Kreis zeigt, können Sie sich für einen `static_cast` :

```
std::cout << "Diameter: " << static_cast<Circle*>(ps)->get_diameter() << std::endl;
```

Das wird den Trick tun. Aber es ist sehr riskant: Wenn `ps` durch etwas anderes als einen `Circle` das Verhalten Ihres Codes undefiniert.

Anstatt russisches Roulette zu spielen, sollten Sie also einen `dynamic_cast` . Dies gilt speziell für polymorphe Klassen:

```
int main() {
    Circle circle(Point(0.0, 0.0), 10);
    Shape &shape = circle;

    std::cout << "The shape has a surface of " << shape.get_surface() << std::endl;

    //shape.get_diameter();   // OUCH !!! Compilation error
```

```
    Circle *pc = dynamic_cast<Circle*>(&shape); // will be nullptr if ps wasn't a circle
    if (pc)
        std::cout << "The shape is a circle of diameter " << pc->get_diameter() << std::endl;
    else
        std::cout << "The shape isn't a circle !" << std::endl;
}
```

Beachten Sie, dass `dynamic_cast` für eine Klasse nicht möglich ist, die nicht polymorph ist. Sie benötigen mindestens eine virtuelle Funktion in der Klasse oder deren Eltern, um sie verwenden zu können.

Polymorphismus und Destruktoren

Wenn eine Klasse polymorph verwendet werden soll und abgeleitete Instanzen als Basiszeiger / -referenzen gespeichert werden sollen, sollte der Destruktor der Basisklasse entweder `virtual` oder `protected` . Im ersten Fall führt dies zur Zerstörung des `vtable` , um die `vtable` zu überprüfen, wobei automatisch der korrekte Destruktor basierend auf dem dynamischen Typ `vtable` wird. Im letzteren Fall ist das Zerstören des Objekts durch einen Basisklassenzeiger / -referenz deaktiviert, und das Objekt kann nur gelöscht werden, wenn es explizit als tatsächlicher Typ behandelt wird.

```
struct VirtualDestructor {
    virtual ~VirtualDestructor() = default;
};

struct VirtualDerived : VirtualDestructor {};

struct ProtectedDestructor {
  protected:
    ~ProtectedDestructor() = default;
};

struct ProtectedDerived : ProtectedDestructor {
    ~ProtectedDerived() = default;
};

// ...

VirtualDestructor* vd = new VirtualDerived;
delete vd; // Looks up VirtualDestructor::~VirtualDestructor() in vtable, sees it's
           // VirtualDerived::~VirtualDerived(), calls that.

ProtectedDestructor* pd = new ProtectedDerived;
delete pd; // Error: ProtectedDestructor::~ProtectedDestructor() is protected.
delete static_cast<ProtectedDerived*>(pd); // Good.
```

Beide Vorgehensweisen garantieren, dass der Destruktor der abgeleiteten Klasse immer für abgeleitete Klasseninstanzen aufgerufen wird, um Speicherverluste zu vermeiden.

Polymorphismus online lesen: https://riptutorial.com/de/cplusplus/topic/1717/polymorphismus

Kapitel 88: Präprozessor

Einführung

Der C-Präprozessor ist ein einfacher Textparser / -ersatz, der vor der eigentlichen Kompilierung des Codes ausgeführt wird. Wird verwendet, um die Sprache C (und später C ++) zu erweitern und zu vereinfachen. Sie kann verwendet werden für:

ein. **Andere Dateien** mit `#include` **einschließen**

b. **Definieren Sie ein Textersetzungsmakro** mit `#define`

c. **Bedingte Kompilierung** mit `#if` `#ifdef`

d. **Plattform- / Compilerspezifische Logik** (als Erweiterung der bedingten Kompilierung)

Bemerkungen

Präprozessoranweisungen werden ausgeführt, bevor Ihre Quelldateien an den Compiler übergeben werden. Sie sind in der Lage, eine bedingte Logik auf sehr niedrigem Niveau auszuführen. Da Präprozessor-Konstrukte (z. B. objektähnliche Makros) nicht wie normale Funktionen eingegeben werden (der Vorverarbeitungsschritt erfolgt vor der Kompilierung), kann der Compiler keine Typprüfung erzwingen. Daher sollten sie sorgfältig verwendet werden.

Examples

Fügen Sie Wachen ein

Eine Headerdatei kann in anderen Headerdateien enthalten sein. Eine Quelldatei (Übersetzungseinheit), die mehrere Header enthält, kann daher indirekt einige Header mehr als einmal enthalten. Wenn eine solche Headerdatei, die mehr als einmal enthalten ist, Definitionen enthält, erkennt der Compiler (nach der Vorverarbeitung) eine Verletzung der One-Definition-Regel (z. B. §3.2 des C ++ - Standards von 2003) und gibt daher eine Diagnose aus, und die Kompilierung schlägt fehl.

Die Verwendung von "Include Guards", die manchmal auch als Header Guards oder Macro Guards bezeichnet werden, wird der Mehrfacheinbeziehung verhindert. Diese werden mit dem Prä - Prozessor implementiert `#define` , `#ifndef` , `#endif` Direktiven.

z.B

```
// Foo.h
#ifndef FOO_H_INCLUDED
#define FOO_H_INCLUDED

class Foo    //  a class definition
```

```
{
};

#endif
```

Der Hauptvorteil der Verwendung von include-Guards besteht darin, dass sie mit allen standardkonformen Compilern und Vorprozessoren zusammenarbeiten.

Include-Guards verursachen jedoch auch Probleme für Entwickler, da sichergestellt werden muss, dass die Makros in allen in einem Projekt verwendeten Kopfzeilen eindeutig sind. Wenn zwei (oder mehr) Header `FOO_H_INCLUDED` als Include-Guard verwenden, verhindert der erste dieser Header, der in einer Kompilierungseinheit enthalten ist, effektiv, dass die anderen Header eingeschlossen werden. Besondere Herausforderungen werden eingeführt, wenn ein Projekt eine Reihe von Bibliotheken von Drittanbietern mit Header-Dateien verwendet, die zufällig gemeinsame Guards enthalten.

Es muss auch sichergestellt werden, dass die in Include-Guards verwendeten Makros keinen Konflikt mit anderen in Header-Dateien definierten Makros verursachen.

Die meisten C ++ - Implementierungen unterstützen auch die `#pragma once` Direktive, mit der sichergestellt wird, dass die Datei nur einmal in einer einzigen Kompilierung enthalten ist. Dies ist eine *De-facto-* Standardrichtlinie , aber sie ist nicht Teil eines ISO-C ++ - Standards. Zum Beispiel:

```
// Foo.h
#pragma once

class Foo
{
};
```

`#pragma once` zwar `#pragma once` einige Probleme mit include - Guards `#pragma once` vermeidet, ist ein `#pragma` - per Definition in den Standards - inhärent ein compilerspezifischer Hook und wird von Compilern, die es nicht unterstützen, ignoriert. Projekte, die `#pragma once` sind schwieriger auf Compiler zu portieren, die dies nicht unterstützen.

Eine Reihe von Codierungsrichtlinien und Sicherheitsstandards für C ++ rät ausdrücklich davon ab, den Präprozessor zu verwenden, mit `#include` von `#include` Include-Header-Dateien oder zur Platzierung von Include-Guards in Headern.

Bedingte Logik und plattformübergreifendes Handling

Kurz gesagt geht es bei der bedingten Vorverarbeitungslogik darum, Codelogik für die Kompilierung mithilfe von Makrodefinitionen verfügbar zu machen oder nicht zur Verfügung zu stellen.

Drei prominente Anwendungsfälle sind:

- verschiedene **App-Profile** (z. B. Debug, Release, Testing, Optimized), die Kandidaten derselben App sein können (z. B. mit zusätzlicher Protokollierung).
- **Plattformübergreifende Kompilierungen** - eine einzige Codebasis, mehrere

Kompilierungsplattformen.
- Verwendung einer gemeinsamen Codebasis für mehrere **Anwendungsversionen** (z. B. Basic-, Premium- und Pro-Versionen einer Software) - mit geringfügig unterschiedlichen Funktionen.

Beispiel a: Plattformübergreifender Ansatz zum Entfernen von Dateien (illustrativ):

```
#ifdef _WIN32
#include <windows.h> // and other windows system files
#endif
#include <cstdio>

bool remove_file(const std::string &path)
{
#ifdef _WIN32
  return DeleteFile(path.c_str());
#elif defined(_POSIX_VERSION) || defined(__unix__)
  return (0 == remove(path.c_str()));
#elif defined(__APPLE__)
  //TODO: check if NSAPI has a more specific function with permission dialog
  return (0 == remove(path.c_str()));
#else
#error "This platform is not supported"
#endif
}
```

Makros wie `_WIN32` , `__APPLE__` oder `__unix__` werden normalerweise durch entsprechende Implementierungen vordefiniert.

Beispiel b: Aktivieren der zusätzlichen Protokollierung für einen Debugbuild:

```
void s_PrintAppStateOnUserPrompt()
{
    std::cout << "--------BEGIN-DUMP---------------\n"
              << AppState::Instance()->Settings().ToString() << "\n"
#if ( 1 == TESTING_MODE ) //privacy: we want user details only when testing
              << ListToString(AppState::UndoStack()->GetActionNames())
              << AppState::Instance()->CrntDocument().Name()
              << AppState::Instance()->CrntDocument().SignatureSHA() << "\n"
#endif
              << "--------END-DUMP---------------\n"
}
```

Beispiel c: Aktivieren Sie ein Premium-Feature in einem separaten Produkt-Build (Hinweis: Dies ist nur zur Veranschaulichung gedacht. Es ist häufig eine bessere Idee, ein Feature freischalten zu lassen, ohne dass eine Anwendung erneut installiert werden muss.)

```
void MainWindow::OnProcessButtonClick()
{
#ifndef _PREMIUM
    CreatePurchaseDialog("Buy App Premium", "This feature is available for our App Premium
users. Click the Buy button to purchase the Premium version at our website");
    return;
#endif
    //...actual feature logic here
```

```
}
```

Einige häufige Tricks:

Symbole zum Aufrufzeitpunkt definieren:

Der Präprozessor kann mit vordefinierten Symbolen (mit optionaler Initialisierung) aufgerufen werden. Zum Beispiel führt dieser Befehl (`gcc -E` nur den Präprozessor aus).

```
gcc -E -DOPTIMISE_FOR_OS_X -DTESTING_MODE=1 Sample.cpp
```

Verarbeitet Sample.cpp auf dieselbe Weise, als wäre dies der `#define OPTIMISE_FOR_OS_X` wenn `#define OPTIMISE_FOR_OS_X` und `#define TESTING_MODE 1` oben in Sample.cpp hinzugefügt würden.

Sicherstellen, dass ein Makro definiert ist:

Wenn ein Makro nicht definiert ist und sein Wert verglichen oder geprüft wird, nimmt der Präprozessor fast immer im Stillen an, dass der Wert `0` . Es gibt einige Möglichkeiten, damit zu arbeiten. Ein Ansatz besteht darin, anzunehmen, dass die Standardeinstellungen als 0 dargestellt werden, und dass Änderungen (z. B. am App-Build-Profil) explizit vorgenommen werden müssen (z. B. ENABLE_EXTRA_DEBUGGING = 0, Satz -DENABLE_EXTRA_DEBUGGING = 1 zum Überschreiben). Ein anderer Ansatz besteht darin, alle Definitionen und Vorgaben explizit zu machen. Dies kann durch eine Kombination der `#ifndef` und `#error` :

```
#ifndef (ENABLE_EXTRA_DEBUGGING)
// please include DefaultDefines.h if not already included.
#    error "ENABLE_EXTRA_DEBUGGING is not defined"
#else
#    if ( 1 == ENABLE_EXTRA_DEBUGGING )
  //code
#    endif
#endif
```

Makros

Makros werden in zwei Hauptgruppen eingeteilt: objektähnliche Makros und funktionsähnliche Makros. Makros werden zu Beginn des Kompilierungsvorgangs als Token-Ersetzung behandelt. Dies bedeutet, dass große (oder sich wiederholende) Codeabschnitte in ein Präprozessormakro abstrahiert werden können.

```
// This is an object-like macro
#define    PI         3.14159265358979

// This is a function-like macro.
// Note that we can use previously defined macros
// in other macro definitions (object-like or function-like)
// But watch out, its quite useful if you know what you're doing, but the
// Compiler doesnt know which type to handle, so using inline functions instead
// is quite recommended (But e.g. for Minimum/Maximum functions it is quite useful)
#define    AREA(r)    (PI*(r)*(r))
```

```
// They can be used like this:
double pi_macro   = PI;
double area_macro = AREA(4.6);
```

Die Qt-Bibliothek verwendet diese Technik, um ein Metaobjektsystem zu erstellen, indem der Benutzer das Makro Q_OBJECT an der Spitze der benutzerdefinierten Klasse, die QObject erweitert, deklariert.

Makronamen werden normalerweise in Großbuchstaben geschrieben, um sie leichter vom normalen Code unterscheiden zu können. Dies ist keine Voraussetzung, sondern wird von vielen Programmierern lediglich als guter Stil betrachtet.

Wenn ein objektartiges Makro gefunden wird, wird es als einfaches Kopieren und Einfügen erweitert, wobei der Name des Makros durch seine Definition ersetzt wird. Wenn ein funktionsähnliches Makro gefunden wird, werden sowohl der Name als auch die Parameter erweitert.

```
double pi_squared = PI * PI;
// Compiler sees:
double pi_squared = 3.14159265358979 * 3.14159265358979;

double area = AREA(5);
// Compiler sees:
double area = (3.14159265358979*(5)*(5))
```

Daher werden funktionsähnliche Makroparameter häufig in Klammern eingeschlossen, wie in `AREA()` oben. Dadurch werden Fehler vermieden, die während der Makroerweiterung auftreten können, insbesondere Fehler, die durch einen einzelnen Makroparameter verursacht werden, der aus mehreren tatsächlichen Werten besteht.

```
#define BAD_AREA(r) PI * r * r

double bad_area = BAD_AREA(5 + 1.6);
// Compiler sees:
double bad_area = 3.14159265358979 * 5 + 1.6 * 5 + 1.6;

double good_area = AREA(5 + 1.6);
// Compiler sees:
double good_area = (3.14159265358979*(5 + 1.6)*(5 + 1.6));
```

Beachten Sie auch, dass aufgrund dieser einfachen Erweiterung die an Makros übergebenen Parameter sorgfältig beachtet werden müssen, um unerwartete Nebenwirkungen zu vermeiden. Wenn der Parameter während der Auswertung geändert wird, wird er jedes Mal geändert, wenn er im erweiterten Makro verwendet wird. Dies ist normalerweise nicht das, was wir wollen. Dies gilt auch, wenn das Makro die Parameter in Klammern einschließt, um zu verhindern, dass die Erweiterung irgendetwas beschädigt.

```
int oops = 5;
double incremental_damage = AREA(oops++);
// Compiler sees:
double incremental_damage = (3.14159265358979*(oops++)*(oops++));
```

Außerdem bieten Makros keine Typsicherheit, was zu schwer verständlichen Fehlern in Bezug auf Typenkonflikte führt.

Da Programmierer normalerweise Zeilen mit einem Semikolon abschließen, werden Makros, die als eigenständige Zeilen verwendet werden sollen, häufig zum "Schlucken" eines Semikolons entworfen. Dies verhindert, dass unbeabsichtigte Fehler durch ein zusätzliches Semikolon verursacht werden.

```
#define IF_BREAKER(Func) Func();

if (some_condition)
    // Oops.
    IF_BREAKER(some_func);
else
    std::cout << "I am accidentally an orphan." << std::endl;
```

In diesem Beispiel unterbricht das unbeabsichtigte Doppel-Semikolon den `if...else` Block und verhindert, dass der Compiler das `else` mit dem `if` übereinstimmt. Um dies zu verhindern, wird das Semikolon aus der Makrodefinition weggelassen, wodurch das Semikolon unmittelbar nach seiner Verwendung "verschluckt" wird.

```
#define IF_FIXER(Func) Func()

if (some_condition)
    IF_FIXER(some_func);
else
    std::cout << "Hooray!  I work again!" << std::endl;
```

Wenn Sie das nachgestellte Semikolon nicht angeben, kann das Makro auch verwendet werden, ohne die aktuelle Anweisung zu beenden, was von Vorteil sein kann.

```
#define DO_SOMETHING(Func, Param) Func(Param, 2)

// ...

some_function(DO_SOMETHING(some_func, 3), DO_SOMETHING(some_func, 42));
```

Normalerweise endet eine Makrodefinition am Ende der Zeile. Wenn ein Makro jedoch mehrere Zeilen abdecken muss, kann am Ende einer Zeile ein Backslash verwendet werden, um dies anzuzeigen. Dieser umgekehrte Schrägstrich muss das letzte Zeichen in der Zeile sein. Dies weist den Präprozessor an, dass die folgende Zeile in der aktuellen Zeile verkettet werden sollte und sie als eine einzige Zeile behandelt. Dies kann mehrmals hintereinander verwendet werden.

```
#define TEXT "I \
am \
many \
lines."

// ...
```

```
std::cout << TEXT << std::endl; // Output:   I am many lines.
```

Dies ist besonders nützlich bei komplexen funktionsähnlichen Makros, die möglicherweise mehrere Zeilen abdecken müssen.

```
#define CREATE_OUTPUT_AND_DELETE(Str) \
    std::string* tmp = new std::string(Str); \
    std::cout << *tmp << std::endl; \
    delete tmp;

// ...

CREATE_OUTPUT_AND_DELETE("There's no real need for this to use 'new'.")
```

Bei komplexeren funktionsartigen Makros kann es nützlich sein, ihnen einen eigenen Bereich zu geben, um mögliche Namenskollisionen zu verhindern oder Objekte am Ende des Makros zu zerstören, ähnlich einer tatsächlichen Funktion. Ein üblicher Ausdruck dafür ist *do while 0* , wobei das Makro in einem *do-while*- Block eingeschlossen ist. Auf diesen Block folgt im Allgemeinen *kein* Semikolon, sodass ein Semikolon verschluckt werden kann.

```
#define DO_STUFF(Type, Param, ReturnVar) do { \
    Type temp(some_setup_values); \
    ReturnVar = temp.process(Param); \
} while (0)

int x;
DO_STUFF(MyClass, 41153.7, x);

// Compiler sees:

int x;
do {
    MyClass temp(some_setup_values);
    x = temp.process(41153.7);
} while (0);
```

Es gibt auch verschiedene Makros. Ähnlich wie bei variadischen Funktionen benötigen diese eine variable Anzahl von Argumenten und erweitern diese dann alle anstelle des speziellen Parameters "Varargs", `__VA_ARGS__` .

```
#define VARIADIC(Param, ...) Param(__VA_ARGS__)

VARIADIC(printf, "%d", 8);
// Compiler sees:
printf("%d", 8);
```

Beachten Sie, dass `__VA_ARGS__` während der Erweiterung an einer beliebigen Stelle in der Definition platziert werden kann und korrekt erweitert wird.

```
#define VARIADIC2(POne, PTwo, PThree, ...) POne(PThree, __VA_ARGS__, PTwo)

VARIADIC2(some_func, 3, 8, 6, 9);
```

```
// Compiler sees:
some_func(8, 6, 9, 3);
```

Im Fall eines variadischen Parameters mit Null-Argumenten behandeln unterschiedliche Compiler das nachfolgende Komma unterschiedlich. Einige Compiler wie Visual Studio schlucken das Komma ohne besondere Syntax. Bei anderen Compilern wie GCC müssen Sie `##` unmittelbar vor `__VA_ARGS__` . Aus diesem Grund ist es ratsam, variadische Makros bedingt zu definieren, wenn die Portabilität ein Problem darstellt.

```
// In this example, COMPILER is a user-defined macro specifying the compiler being used.

#if       COMPILER == "VS"
    #define VARIADIC3(Name, Param, ...) Name(Param, __VA_ARGS__)
#elif     COMPILER == "GCC"
    #define VARIADIC3(Name, Param, ...) Name(Param, ##__VA_ARGS__)
#endif /* COMPILER */
```

Preprozessor-Fehlermeldungen

Kompilierungsfehler können mit dem Präprozessor generiert werden. Dies ist aus einer Reihe von Gründen nützlich, zu denen einige gehören: Benachrichtigen eines Benutzers, wenn er sich auf einer nicht unterstützten Plattform oder einem nicht unterstützten Compiler befindet

zB Rückgabefehler, wenn die gcc-Version 3.0.0 oder früher ist.

```
#if __GNUC__ < 3
#error "This code requires gcc > 3.0.0"
#endif
```

zB Fehler beim Kompilieren auf einem Apple-Computer.

```
#ifdef __APPLE__
#error "Apple products are not supported in this release"
#endif
```

Vordefinierte Makros

Vordefinierte Makros sind diejenigen, die der Compiler definiert (im Gegensatz zu denen, die der Benutzer in der Quelldatei definiert). Diese Makros dürfen vom Benutzer nicht neu definiert oder undefiniert werden.

Die folgenden Makros sind durch den C ++ - Standard vordefiniert:

- `__LINE__` enthält die Zeilennummer der Zeile dieses Makro auf verwendet wird, und kann durch die geändert werden `#line` Richtlinie.
- `__FILE__` enthält den Dateinamen der Datei dieses Makro in verwendet wird, und kann durch die geändert werden `#line` Richtlinie.
- `__DATE__` enthält das Datum (im Format `"Mmm dd yyyy"`) der `"Mmm dd yyyy"` , wobei *Mmm* so formatiert ist, als wäre es durch einen Aufruf von `std::asctime()` .
- `__TIME__`

enthält die Zeit (im Format `"hh:mm:ss"`) der `__TIME__` .
- `__cplusplus` wird durch (konforme) C ++ - Compiler beim Kompilieren von C ++ - Dateien definiert. Sein Wert ist die Standardversion, mit der der Compiler **vollständig** konform ist, dh `199711L` für C ++ 98 und C ++ 03, `201103L` für C ++ 11 und `201402L` für C ++ 14-Standard.

c ++ 11

- `__STDC_HOSTED__` ist auf `1` festgelegt, wenn die Implementierung *gehostet wird* , oder `0` wenn sie *freistehend ist* .

c ++ 17

- `__STDCPP_DEFAULT_NEW_ALIGNMENT__` enthält ein literal `size_t` , das die Ausrichtung ist, die für einen Aufruf des `__STDCPP_DEFAULT_NEW_ALIGNMENT__` `operator new` .

Außerdem dürfen die folgenden Makros durch Implementierungen vordefiniert werden und sind möglicherweise vorhanden:

- `__STDC__` hat eine implementierungsabhängige Bedeutung und wird normalerweise nur beim Kompilieren einer Datei als C definiert, um die vollständige Konformität mit dem C-Standard anzuzeigen. (Oder niemals, wenn der Compiler dieses Makro nicht unterstützt.)

c ++ 11

- `__STDC_VERSION__` hat eine implementierungsabhängige Bedeutung und der Wert ist normalerweise die C-Version, ähnlich wie `__cplusplus` die C ++ - Version ist. (Oder ist nicht einmal definiert, wenn der Compiler dieses Makro nicht unterstützt.)
- `__STDC_MB_MIGHT_NEQ_WC__` ist auf `1` definiert, wenn die Werte der engen Kodierung des `__STDC_MB_MIGHT_NEQ_WC__` möglicherweise nicht den Werten ihrer breiten Entsprechungen entsprechen (z. B. if `(uintmax_t)'x' != (uintmax_t)L'x'`)
- `__STDC_ISO_10646__` ist definiert, wenn `wchar_t` als Unicode codiert ist, und erweitert sich zu einer Ganzzahlkonstante in der Form `yyyymmL` , die die letzte unterstützte Unicode- `yyyymmL` angibt.
- `__STDCPP_STRICT_POINTER_SAFETY__` ist auf `1` definiert, wenn die Implementierung *strikte* `__STDCPP_STRICT_POINTER_SAFETY__` (`__STDCPP_STRICT_POINTER_SAFETY__` ist die `__STDCPP_STRICT_POINTER_SAFETY__` *gelockert*)
- `__STDCPP_THREADS__` ist `1` , wenn das Programm mehr als einen Ausführungsthread haben kann (gilt für *freistehende Implementierungen* - *gehostete Implementierungen* können immer mehr als einen Thread haben)

Erwähnenswert ist auch `__func__` , bei dem es sich nicht um ein Makro, sondern um eine vordefinierte funktionslokale Variable handelt. Es enthält den Namen der Funktion, in der es verwendet wird, als statisches Zeichenarray in einem implementierungsdefinierten Format.

Zusätzlich zu diesen standardmäßigen vordefinierten Makros können Compiler über einen eigenen Satz vordefinierter Makros verfügen. Um diese zu lernen, muss man sich auf die Dokumentation des Compilers beziehen. Z.B:

- gcc

- Microsoft Visual C ++
- klirren
- Intel C ++ Compiler

Einige Makros dienen lediglich der Abfrage einiger Funktionen:

```
#ifdef __cplusplus // if compiled by C++ compiler
extern "C"{ // C code has to be decorated
   // C library header declarations here
}
#endif
```

Andere sind sehr nützlich für das Debuggen:

c ++ 11

```
bool success = doSomething( /*some arguments*/ );
if( !success ){
    std::cerr << "ERROR: doSomething() failed on line " << __LINE__ - 2
              << " in function " << __func__ << "()"
              << " in file " << __FILE__
              << std::endl;
}
```

Und andere zur trivialen Versionskontrolle:

```
int main( int argc, char *argv[] ){
    if( argc == 2 && std::string( argv[1] ) == "-v" ){
        std::cout << "Hello World program\n"
                  << "v 1.1\n" // I have to remember to update this manually
                  << "compiled: " << __DATE__ << ' ' << __TIME__ // this updates automagically
                  << std::endl;
    }
    else{
        std::cout << "Hello World!\n";
    }
}
```

X-Makros

Eine idiomatische Technik zum Erzeugen sich wiederholender Codestrukturen zur Kompilierzeit.

Ein X-Makro besteht aus zwei Teilen: der Liste und der Ausführung der Liste.

Beispiel:

```
#define LIST \
    X(dog)   \
    X(cat)   \
    X(racoon)

// class Animal {
//  public:
//    void say();
```

```
// };

#define X(name) Animal name;
LIST
#undef X

int main() {
#define X(name) name.say();
    LIST
#undef X

    return 0;
}
```

welches vom Präprozessor in folgendes erweitert wird:

```
Animal dog;
Animal cat;
Animal racoon;

int main() {
    dog.say();
    cat.say();
    racoon.say();

    return 0;
}
```

Da Listen größer werden (sagen wir mehr als 100 Elemente), hilft diese Technik, übermäßiges Kopieren und Einfügen zu vermeiden.

Quelle: https://en.wikipedia.org/wiki/X_Macro

Siehe auch: X-Makros

Wenn Sie ein nahtlos irrelevantes `X` vor der Verwendung von `LIST` nicht definieren möchten, können Sie auch einen Makronamen als Argument übergeben:

```
#define LIST(MACRO) \
    MACRO(dog) \
    MACRO(cat) \
    MACRO(racoon)
```

Nun legen Sie explizit fest, welches Makro beim Erweitern der Liste verwendet werden soll, z

```
#define FORWARD_DECLARE_ANIMAL(name) Animal name;
LIST(FORWARD_DECLARE_ANIMAL)
```

Wenn für jeden Aufruf des `MACRO` zusätzliche Parameter erforderlich sind - konstant in Bezug auf die Liste, können variadische Makros verwendet werden

```
//a walkaround for Visual studio
#define EXPAND(x) x
```

```
#define LIST(MACRO, ...) \
    EXPAND(MACRO(dog, __VA_ARGS__)) \
    EXPAND(MACRO(cat, __VA_ARGS__)) \
    EXPAND(MACRO(racoon, __VA_ARGS__))
```

Das erste Argument wird von der `LIST` bereitgestellt, während der Rest vom Benutzer beim Aufruf der `LIST` bereitgestellt wird. Zum Beispiel:

```
#define FORWARD_DECLARE(name, type, prefix) type prefix##name;
LIST(FORWARD_DECLARE,Animal,anim_)
LIST(FORWARD_DECLARE,Object,obj_)
```

wird zu erweitern

```
Animal anim_dog;
Animal anim_cat;
Animal anim_racoon;
Object obj_dog;
Object obj_cat;
Object obj_racoon;
```

#pragma einmal

Die meisten, aber nicht alle C ++ - Implementierungen unterstützen die `#pragma once` Direktive, mit der sichergestellt wird, dass die Datei nur einmal in einer einzigen Kompilierung enthalten ist. Es ist nicht Teil eines ISO C ++ - Standards. Zum Beispiel:

```
// Foo.h
#pragma once

class Foo
{
};
```

`#pragma once` zwar `#pragma once` einige Probleme mit include - Guards `#pragma once` vermeidet, ist ein `#pragma` - per Definition in den Standards - inhärent ein compilerspezifischer Hook und wird von Compilern, die es nicht unterstützen, ignoriert. Projekte, die `#pragma once` müssen modifiziert werden, um standardkonform zu sein.

Bei einigen Compilern - insbesondere bei solchen, die vorkompilierte Header verwenden - kann `#pragma once` zu einer erheblichen Beschleunigung des Kompilierungsprozesses führen. In ähnlicher Weise erzielen einige Vorprozessoren eine Beschleunigung der Kompilierung, indem sie nachverfolgen, welche Kopfzeilen über Wachen verfügen. Der Nettonutzen, wenn sowohl `#pragma once` als auch `#pragma once` verwendet werden, hängt von der Implementierung ab und kann entweder eine Erhöhung oder Verkürzung der Kompilierungszeiten darstellen.

`#pragma once` Kombination mit Include Guards das empfohlene Layout für Headerdateien beim Schreiben von MFC-basierten Anwendungen unter Windows. Es wurde von Visual Studio's `add class` `add dialog` `add windows` . Daher ist es sehr üblich, sie in C ++ Windows-Antragstellern

kombiniert zu finden.

Präprozessoroperatoren

`#` Operator `#` oder der String-Operator wird verwendet, um einen Makro-Parameter in ein String-Literal zu konvertieren. Es kann nur mit Makros verwendet werden, die Argumente haben.

```
// preprocessor will convert the parameter x to the string literal x
#define PRINT(x) printf(#x "\n")

PRINT(This line will be converted to string by preprocessor);
// Compiler sees
printf("This line will be converted to string by preprocessor""\n");
```

Der Compiler verkettet zwei Zeichenfolgen, und das letzte Argument `printf()` ist ein Zeichenfolgenliteral mit einem Zeilenvorschubzeichen am Ende.

Der Präprozessor ignoriert die Leerzeichen vor oder nach dem Makroargument. Die folgende Druckaussage liefert also dasselbe Ergebnis.

```
PRINT(   This line will be converted to string by preprocessor );
```

Wenn für den Parameter des Zeichenfolgenlitals eine Escape-Sequenz wie vor einem doppelten Anführungszeichen () erforderlich ist, wird er automatisch vom Präprozessor eingefügt.

```
PRINT(This "line" will be converted to "string" by preprocessor);
// Compiler sees
printf("This \"line\" will be converted to \"string\" by preprocessor""\n");
```

`##` Operator oder Token-Einfügeoperator wird verwendet, um zwei Parameter oder Token eines Makros zu verketten.

```
// preprocessor will combine the variable and the x
#define PRINT(x) printf("variable" #x " = %d", variable##x)

int variableY = 15;
PRINT(Y);
//compiler sees
printf("variable""Y"" = %d", variableY);
```

und die endgültige Ausgabe wird sein

```
variableY = 15
```

Präprozessor online lesen: https://riptutorial.com/de/cplusplus/topic/1098/praprozessor

Kapitel 89: Profilierung

Examples

Profilierung mit gcc und gprof

Mit dem GNU gprof-Profiler gprof können Sie Ihren Code profilieren. Um es zu verwenden, müssen Sie die folgenden Schritte ausführen:

1. Erstellen Sie die Anwendung mit Einstellungen zum Generieren von Profilierungsinformationen
2. Generieren Sie Informationen zur Profilerstellung, indem Sie die erstellte Anwendung ausführen
3. Zeigen Sie die generierten Profilinformationen mit gprof an

Um die Anwendung mit Einstellungen zum Generieren von Profilierungsinformationen zu erstellen, fügen Sie das Flag `-pg` . So könnten wir zum Beispiel verwenden

```
$ gcc -pg *.cpp -o app
```

oder

```
$ gcc -O2 -pg *.cpp -o app
```

und so weiter.

Sobald die Anwendung, beispielsweise die `app` , erstellt wurde, führen Sie sie wie gewohnt aus:

```
$ ./app
```

Dies sollte eine Datei namens `gmon.out` .

Um die Ergebnisse der Profilerstellung zu sehen, führen Sie jetzt aus

```
$ gprof app gmon.out
```

(Beachten Sie, dass wir sowohl die Anwendung als auch die generierte Ausgabe bereitstellen).

Natürlich können Sie auch pfeifen oder umleiten:

```
$ gprof app gmon.out | less
```

und so weiter.

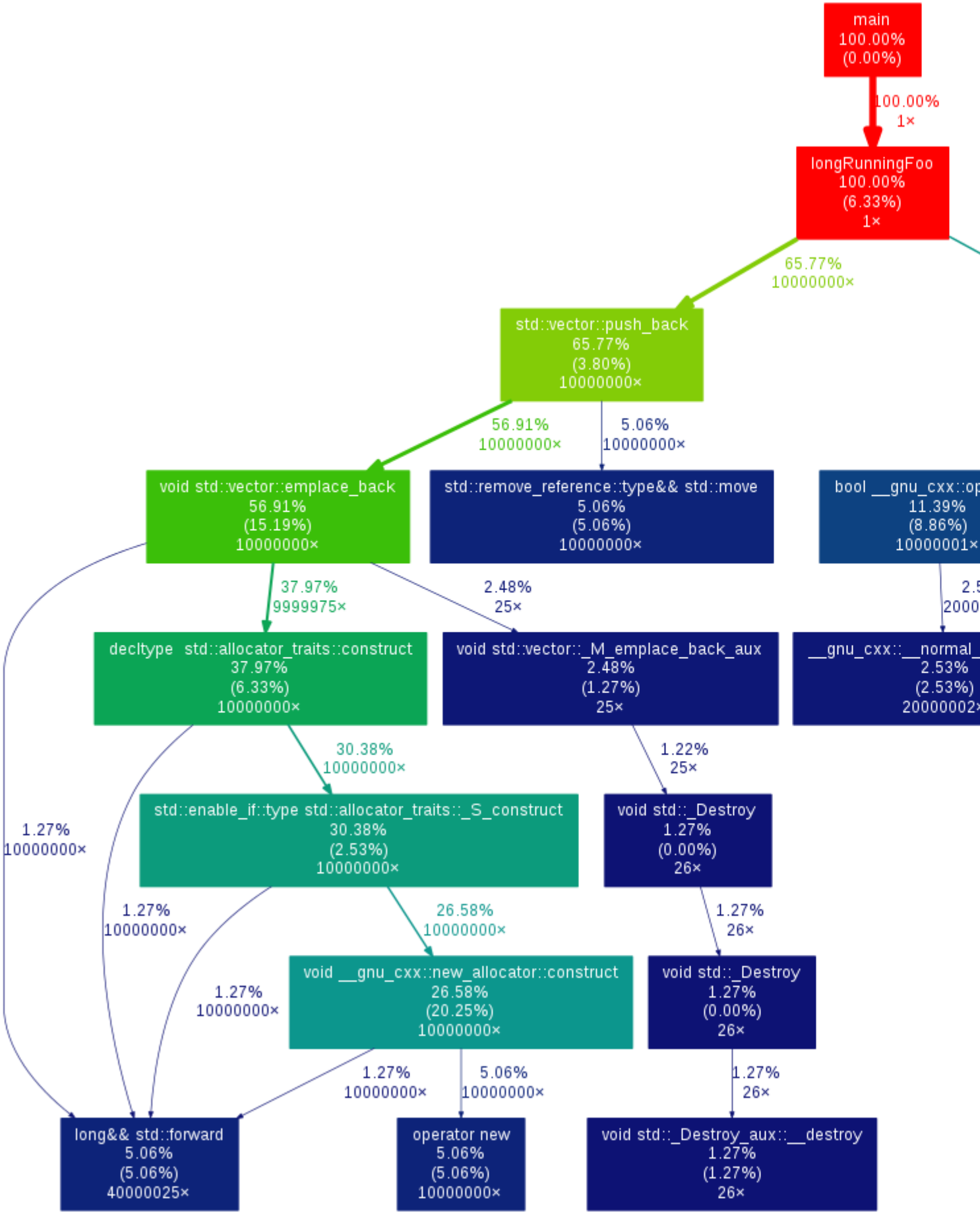
Das Ergebnis des letzten Befehls sollte eine Tabelle sein, deren Zeilen die Funktionen sind, und deren Spalten die Anzahl der Aufrufe, die Gesamtzeit und die selbst verbrachte Zeit (dh die Zeit in der Funktion, die Aufrufe von Kindern enthält) angeben.

Callgraph-Diagramme mit gperf2dot erstellen

Für komplexere Anwendungen kann es schwierig sein, flache Ausführungsprofile zu verfolgen. Aus diesem Grund generieren viele Tools zur Profilerstellung auch eine Art kommentierter Callgraph-Informationen.

gperf2dot konvertiert Textausgaben vieler Profiler (Linux perf, callgrind, oprofile usw.) in ein Callgraph-Diagramm. Sie können es verwenden, indem Sie Ihren Profiler `gprof` (Beispiel für `gprof`):

```
# compile with profiling flags
g++ *.cpp -pg
# run to generate profiling data
./main
# translate profiling data to text, create image
gprof ./main | gprof2dot -s | dot -Tpng -o output.png
```



Profilierung der CPU-Nutzung mit gcc und Google Perf Tools

Google Perf Tools bietet auch einen CPU-Profiler mit einer etwas freundlicheren Benutzeroberfläche. Um es zu benutzen:

1. Installieren Sie die Google Perf Tools
2. Kompilieren Sie Ihren Code wie gewohnt
3. Fügen Sie die Bibliothek `libprofiler` zur Laufzeit Ihrem Bibliotheksladepfad hinzu
4. Verwenden Sie `pprof` , um ein Profil für die flache Ausführung oder ein Callgraph-Diagramm zu erstellen

Zum Beispiel:

```
# compile code
g++ -O3 -std=c++11 main.cpp -o main

# run with profiler
LD_PRELOAD=/usr/local/lib/libprofiler.so CPUPROFILE=main.prof CPUPROFILE_FREQUENCY=100000
./main
```

woher:

- `CPUPROFILE` gibt die Ausgabedatei zum Profilieren von Daten an
- `CPUPROFILE_FREQUENCY` gibt die Abtastfrequenz des Profilers an.

Verwenden Sie `pprof` , um die Profilierungsdaten `pprof` .

Sie können ein Flat-Call-Profil als Text generieren:

```
$ pprof --text ./main main.prof
PROFILE: interrupts/evictions/bytes = 67/15/2016
pprof --text --lines ./main main.prof
Using local file ./main.
Using local file main.prof.
Total: 67 samples
      22  32.8%  32.8%       67 100.0% longRunningFoo ??:0
      20  29.9%  62.7%       20  29.9% __memmove_ssse3_back /build/eglibc-3GlaMS/eglibc-
2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1627
       4   6.0%  68.7%        4   6.0% __memmove_ssse3_back /build/eglibc-3GlaMS/eglibc-
2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1619
       3   4.5%  73.1%        3   4.5% __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:388
       3   4.5%  77.6%        3   4.5% __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:401
       2   3.0%  80.6%        2   3.0% __munmap /build/eglibc-3GlaMS/eglibc-
2.19/misc/../sysdeps/unix/syscall-template.S:81
       2   3.0%  83.6%       12  17.9% __random /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random.c:298
       2   3.0%  86.6%        2   3.0% __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:385
       2   3.0%  89.6%        2   3.0% rand /build/eglibc-3GlaMS/eglibc-2.19/stdlib/rand.c:26
       1   1.5%  91.0%        1   1.5% __memmove_ssse3_back /build/eglibc-3GlaMS/eglibc-
2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1617
       1   1.5%  92.5%        1   1.5% __memmove_ssse3_back /build/eglibc-3GlaMS/eglibc-
2.19/string/../sysdeps/x86_64/multiarch/memcpy-ssse3-back.S:1623
       1   1.5%  94.0%        1   1.5% __random /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random.c:293
       1   1.5%  95.5%        1   1.5% __random /build/eglibc-3GlaMS/eglibc-
```

```
2.19/stdlib/random.c:296
       1   1.5%  97.0%        1   1.5% __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:371
       1   1.5%  98.5%        1   1.5% __random_r /build/eglibc-3GlaMS/eglibc-
2.19/stdlib/random_r.c:381
       1   1.5% 100.0%        1   1.5% rand /build/eglibc-3GlaMS/eglibc-2.19/stdlib/rand.c:28
       0   0.0% 100.0%       67 100.0% __libc_start_main /build/eglibc-3GlaMS/eglibc-
2.19/csu/libc-start.c:287
       0   0.0% 100.0%       67 100.0% _start ??:0
       0   0.0% 100.0%       67 100.0% main ??:0
       0   0.0% 100.0%       14  20.9% rand /build/eglibc-3GlaMS/eglibc-2.19/stdlib/rand.c:27
       0   0.0% 100.0%       27  40.3% std::vector::_M_emplace_back_aux ??:0
```

... oder Sie können eine kommentierte Callgraph-Datei in einer PDF-Datei erzeugen mit:

```
pprof --pdf ./main main.prof > out.pdf
```

Profilierung online lesen: https://riptutorial.com/de/cplusplus/topic/5347/profilierung

Kapitel 90: RAII: Ressourcenakquisition ist Initialisierung

Bemerkungen

RAII steht für **R** eSource Ü bernahme **I** s **I** nitialization. RAII, gelegentlich auch als SBRM (Scope-Based Resource Management) oder RRID (Resource Release Is Destruction) bezeichnet, ist ein Idiom, mit dem Ressourcen an die Lebensdauer des Objekts gebunden werden. In C ++ wird der Destruktor für ein Objekt immer ausgeführt, wenn ein Objekt seinen Gültigkeitsbereich verlässt - wir können dies nutzen, um die Ressourcenbereinigung mit der Objektzerstörung zu verknüpfen.

Jedes Mal, wenn Sie eine Ressource (z. B. eine Sperre, ein Dateihandle, einen zugewiesenen Puffer) benötigen, die Sie eventuell freigeben müssen, sollten Sie ein Objekt zur Verwaltung dieser Ressourcenverwaltung in Betracht ziehen. Das Abwickeln des Stapels wird unabhängig von der Ausnahme oder dem vorzeitigen Beenden des Bereichs ausgeführt. Daher bereinigt das Ressourcenhandlerobjekt die Ressource für Sie, ohne dass Sie alle möglichen aktuellen und zukünftigen Codepfade sorgfältig prüfen müssen.

Es ist erwähnenswert, dass RAII den Entwickler nicht völlig frei macht, über die Lebensdauer von Ressourcen nachzudenken. Ein Fall ist offensichtlich ein Absturz- oder Beendigungsaufruf (), der den Aufruf von Destruktoren verhindert. Da das Betriebssystem nach dem Beenden eines Prozesses prozesslokale Ressourcen wie Speicher bereinigt, ist dies in den meisten Fällen kein Problem. Bei Systemressourcen (z. B. Named Pipes, Sperrdateien, Shared Memory) benötigen Sie jedoch noch Einrichtungen, um den Fall zu bewältigen, dass ein Prozess nicht nach sich selbst bereinigt wird, dh beim Starttest, ob die Sperrdatei vorhanden ist. Überprüfen Sie, ob der Prozess tatsächlich vorhanden ist, und handeln Sie entsprechend.

Eine andere Situation ist, wenn ein Unix-Prozess eine Funktion aus der Exec-Familie aufruft, dh nach einem Fork-Exec, um einen neuen Prozess zu erstellen. Der untergeordnete Prozess verfügt über eine vollständige Kopie des übergeordneten Speichers (einschließlich der RAII-Objekte). Sobald exec jedoch aufgerufen wurde, wird keiner der Destruktoren in diesem Prozess aufgerufen. Wenn dagegen ein Prozess verzweigt ist und keiner der Prozesse exec aufgerufen hat, werden alle Ressourcen in beiden Prozessen bereinigt. Dies gilt nur für alle Ressourcen, die tatsächlich in der Verzweigung dupliziert wurden. Bei Systemressourcen haben beide Prozesse jedoch nur einen Verweis auf die Ressource (dh den Pfad zu einer Sperrdatei). Beide versuchen, sie einzeln freizugeben, was möglicherweise dazu führt der andere Prozess zum Scheitern verurteilt.

Examples

Sperren

Schlechte Verriegelung:

```
std::mutex mtx;

void bad_lock_example() {
    mtx.lock();
    try
    {
        foo();
        bar();
        if (baz()) {
            mtx.unlock();   // Have to unlock on each exit point.
            return;
        }
        quux();
        mtx.unlock();       // Normal unlock happens here.
    }
    catch(...) {
        mtx.unlock();       // Must also force unlock in the presence of
        throw;              // exceptions and allow the exception to continue.
    }
}
```

Dies ist der falsche Weg, um das Verriegeln und Entriegeln des Mutex zu implementieren. Um die korrekte Freigabe des Mutex mit `unlock()` sicherzustellen, muss der Programmierer sicherstellen, dass alle Flüsse, die zum Beenden der Funktion führen, einen Aufruf von `unlock()` . Wie oben gezeigt, ist dies ein spröder Prozess, da alle Betreuer dem Muster manuell folgen müssen.

Bei Verwendung einer entsprechend gestalteten Klasse zur Implementierung von RAII ist das Problem trivial:

```
std::mutex mtx;

void good_lock_example() {
    std::lock_guard<std::mutex> lk(mtx);   // constructor locks.
                                           // destructor unlocks. destructor call
                                           // guaranteed by language.
    foo();
    bar();
    if (baz()) {
        return;
    }
    quux();
}
```

`lock_guard` ist eine extrem einfache Klassenvorlage, die einfach `lock()` für ihr Argument in ihrem Konstruktor aufruft, einen Verweis auf das Argument `lock_guard` `unlock()` für das Argument in seinem Destruktor aufruft. Das heißt, wenn der `lock_guard` den `lock_guard` , ist die `lock_guard` des `mutex` garantiert. Es spielt keine Rolle, ob der Ausnahmefall eine Ausnahme oder eine vorzeitige Rückgabe ist - alle Fälle werden behandelt. Unabhängig vom Kontrollfluss haben wir garantiert, dass wir die Entriegelung korrekt durchführen.

Schließlich / ScopeExit

Für den Fall, dass wir keine speziellen Klassen schreiben wollen, um eine Ressource zu behandeln, schreiben wir möglicherweise eine generische Klasse:

```
template<typename Function>
class Finally final
{
public:
    explicit Finally(Function f) : f(std::move(f)) {}
    ~Finally() { f(); } // (1) See below

    Finally(const Finally&) = delete;
    Finally(Finally&&) = default;
    Finally& operator =(const Finally&) = delete;
    Finally& operator =(Finally&&) = delete;
private:
    Function f;
};
// Execute the function f when the returned object goes out of scope.
template<typename Function>
auto onExit(Function &&f) { return Finally<std::decay_t<Function>>{std::forward<Function>(f)};
}
```

Und sein Beispielgebrauch

```
void foo(std::vector<int>& v, int i)
{
    // ...

    v[i] += 42;
    auto autoRollBackChange = onExit([&](){ v[i] -= 42; });

    // ... code as recursive call `foo(v, i + 1)`
}
```

Anmerkung (1): Einige Diskussionen über die Definition von Destruktoren müssen berücksichtigt werden, um eine Ausnahme zu behandeln:

- `~Finally() noexcept { f(); }` : `std::terminate` wird im Ausnahmefall aufgerufen
- `~Finally() noexcept(noexcept(f())) { f(); }` : terminate () wird nur im Ausnahmefall beim Stack-Abwickeln aufgerufen.
- `~Finally() noexcept { try { f(); } catch (...) { /* ignore exception (might log it) */} }` Kein `std::terminate` aufgerufen, aber Fehler können nicht behandelt werden (auch nicht für das Abwickeln ohne Stack).

ScopeSuccess (c ++ 17)

C ++ 17

Dank `int std::uncaught_exceptions()` können wir eine Aktion implementieren, die nur bei Erfolg ausgeführt wird (keine ausgelöste Ausnahme im Gültigkeitsbereich). Bisher konnte `bool std::uncaught_exception()` nur feststellen, ob **ein** Stack-Abwickelvorgang ausgeführt wird.

```
#include <exception>
#include <iostream>

template <typename F>
class ScopeSuccess
```

```
{
private:
    F f;
    int uncaughtExceptionCount = std::uncaught_exceptions();
public:
    explicit ScopeSuccess(const F& f) : f(f) {}
    ScopeSuccess(const ScopeSuccess&) = delete;
    ScopeSuccess& operator =(const ScopeSuccess&) = delete;

    // f() might throw, as it can be caught normally.
    ~ScopeSuccess() noexcept(noexcept(f())) {
        if (uncaughtExceptionCount == std::uncaught_exceptions()) {
            f();
        }
    }
};

struct Foo {
    ~Foo() {
        try {
            ScopeSuccess logSuccess{[](){std::cout << "Success 1\n";}};
            // Scope succeeds,
            // even if Foo is destroyed during stack unwinding
            // (so when 0 < std::uncaught_exceptions())
            // (or previously std::uncaught_exception() == true)
        } catch (...) {
        }
        try {
            ScopeSuccess logSuccess{[](){std::cout << "Success 2\n";}};

            throw std::runtime_error("Failed"); // returned value
                                                // of std::uncaught_exceptions increases
        } catch (...) { // returned value of std::uncaught_exceptions decreases
        }
    }

};

int main()
{
    try {
        Foo foo;

        throw std::runtime_error("Failed"); // std::uncaught_exceptions() == 1
    } catch (...) { // std::uncaught_exceptions() == 0
    }
}
```

Ausgabe:

```
Success 1
```

ScopeFail (c ++ 17)

C ++ 17

Dank `int std::uncaught_exceptions()` können wir eine Aktion implementieren, die nur bei einem Fehler ausgeführt wird (Ausnahme im Geltungsbereich ausgelöst). Bisher konnte `bool`

`std::uncaught_exception()` nur feststellen, ob **ein** Stack-Abwickelvorgang ausgeführt wird.

```
#include <exception>
#include <iostream>

template <typename F>
class ScopeFail
{
private:
    F f;
    int uncaughtExceptionCount = std::uncaught_exceptions();
public:
    explicit ScopeFail(const F& f) : f(f) {}
    ScopeFail(const ScopeFail&) = delete;
    ScopeFail& operator =(const ScopeFail&) = delete;

    // f() should not throw, else std::terminate is called.
    ~ScopeFail() {
        if (uncaughtExceptionCount != std::uncaught_exceptions()) {
            f();
        }
    }
};

struct Foo {
    ~Foo() {
        try {
            ScopeFail logFailure{[](){std::cout << "Fail 1\n";}};
            // Scope succeeds,
            // even if Foo is destroyed during stack unwinding
            // (so when 0 < std::uncaught_exceptions())
            // (or previously std::uncaught_exception() == true)
        } catch (...) {
        }
        try {
            ScopeFail logFailure{[](){std::cout << "Failure 2\n";}};

            throw std::runtime_error("Failed"); // returned value
                                                // of std::uncaught_exceptions increases
        } catch (...) { // returned value of std::uncaught_exceptions decreases
        }
    }

};

int main()
{
    try {
        Foo foo;

        throw std::runtime_error("Failed"); // std::uncaught_exceptions() == 1
    } catch (...) { // std::uncaught_exceptions() == 0
    }
}
```

Ausgabe:

```
Failure 2
```

RAII: Ressourcenakquisition ist Initialisierung online lesen:
https://riptutorial.com/de/cplusplus/topic/1320/raii--ressourcenakquisition-ist-initialisierung

Kapitel 91: Refactoring-Techniken

Einführung

Refactoring bezieht sich auf die Änderung von vorhandenem Code in eine verbesserte Version. Obwohl Refactoring häufig durchgeführt wird, während Code geändert wird, um Features hinzuzufügen oder Fehler zu beheben, bezieht sich der Begriff insbesondere auf die Verbesserung von Code, ohne notwendigerweise Features hinzuzufügen oder Fehler zu beheben.

Examples

Refactoring durchlaufen

Hier ist ein Programm, das vom Refactoring profitieren könnte. Es ist ein einfaches Programm unter Verwendung von C ++ 11, das alle Primzahlen von 1 bis 100 berechnen und drucken soll und auf einem Programm basiert, das zur Überprüfung in CodeReview veröffentlicht wurde .

```
#include <iostream>
#include <vector>
#include <cmath>

int main()
{
    int l = 100;
    bool isprime;
    std::vector<int> primes;
    primes.push_back(2);
    for (int no = 3; no < l; no += 2) {
        isprime = true;
        for (int primecount=0; primes[primecount] <= std::sqrt(no); ++primecount) {
            if (no % primes[primecount] == 0) {
                isprime = false;
                break;
            } else if (primes[primecount] * primes[primecount] > no) {
                std::cout << no << "\n";
                break;
            }
        }
        if (isprime) {
            std::cout << no << " ";
            primes.push_back(no);
        }
    }
    std::cout << "\n";
}
```

Die Ausgabe dieses Programms sieht folgendermaßen aus:

    3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

Das erste, was wir bemerken, ist, dass das Programm `2` nicht druckt, was eine Primzahl ist. Wir

könnten einfach eine Zeile Code fügen Sie einfach ohne Änderung der Rest des Programms , dass eine Konstante zu drucken, aber es könnte sauberer sein , das Programm *Refactoring* es in zwei Teile zu spalten - eine , die die Primzahl - Liste eine andere schafft , die sie druckt . So könnte das aussehen:

```
#include <iostream>
#include <vector>
#include <cmath>

std::vector<int> prime_list(int limit)
{
    bool isprime;
    std::vector<int> primes;
    primes.push_back(2);
    for (int no = 3; no < limit; no += 2) {
        isprime = true;
        for (int primecount=0; primes[primecount] <= std::sqrt(no); ++primecount) {
            if (no % primes[primecount] == 0) {
                isprime = false;
                break;
            } else if (primes[primecount] * primes[primecount] > no) {
                break;
            }
        }
        if (isprime) {
            primes.push_back(no);
        }
    }
    return primes;
}

int main()
{
    std::vector<int> primes = prime_list(100);
    for (std::size_t i = 0; i < primes.size(); ++i) {
        std::cout << primes[i] << ' ';
    }
    std::cout << '\n';
}
```

Beim Versuch dieser Version sehen wir, dass sie jetzt richtig funktioniert:

    2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97

Der nächste Schritt ist zu bemerken, dass die zweite `if` Klausel nicht wirklich benötigt wird. Die Logik in der Schleife sucht nach Primfaktoren von jeder gegebenen Zahl bis zur Quadratwurzel dieser Zahl. Dies funktioniert, weil bei Primfaktoren einer Zahl mindestens einer von ihnen kleiner oder gleich der Quadratwurzel dieser Zahl sein muss. Wenn Sie nur diese Funktion überarbeiten (der Rest des Programms bleibt gleich), erhalten Sie dieses Ergebnis:

```
std::vector<int> prime_list(int limit)
{
    bool isprime;
    std::vector<int> primes;
    primes.push_back(2);
    for (int no = 3; no < limit; no += 2) {
```

```
        isprime = true;
        for (int primecount=0; primes[primecount] <= std::sqrt(no); ++primecount) {
            if (no % primes[primecount] == 0) {
                isprime = false;
                break;
            }
        }
        if (isprime) {
            primes.push_back(no);
        }
    }
    return primes;
}
```

Wir können noch weiter gehen und Variablennamen ändern, um sie etwas beschreibender zu gestalten. Zum Beispiel ist `primecount` nicht wirklich eine Anzahl von Primzahlen. Stattdessen handelt es sich um eine Indexvariable in den Vektor bekannter Primzahlen. Auch wenn `no` manchmal als Abkürzung für "number" verwendet wird, ist es beim mathematischen Schreiben üblicher, `n` zu verwenden. Wir können auch einige Modifikationen vornehmen, indem wir die `break` beseitigen und Variablen näher am Verwendungsort deklarieren.

```
std::vector<int> prime_list(int limit)
{
    std::vector<int> primes{2};
    for (int n = 3; n < limit; n += 2) {
        bool isprime = true;
        for (int i=0; isprime && primes[i] <= std::sqrt(n); ++i) {
            isprime &= (n % primes[i] != 0);
        }
        if (isprime) {
            primes.push_back(n);
        }
    }
    return primes;
}
```

Wir können `main` auch umgestalten, um ein "Range-for" zu verwenden, um es etwas netter zu machen:

```
int main()
{
    std::vector<int> primes = prime_list(100);
    for (auto p : primes) {
        std::cout << p << ' ';
    }
    std::cout << '\n';
}
```

Dies ist nur eine Möglichkeit, Refactoring durchzuführen. Andere können andere Entscheidungen treffen. Der Zweck des Refactorings bleibt jedoch derselbe, dh die Lesbarkeit und möglicherweise die Leistung des Codes müssen verbessert werden, ohne dass Features hinzugefügt werden müssen.

Gehe zu Bereinigung

In C ++ - Codebasen, die früher C waren, kann man das Muster `goto cleanup` . Da der Befehl `goto` den Workflow einer Funktion schwieriger zu verstehen macht, wird dies häufig vermieden. Häufig kann es durch return-Anweisungen, Schleifen und Funktionen ersetzt werden. Allerdings muss man mit der `goto cleanup` die `goto cleanup` loswerden.

```
short calculate(VectorStr **data) {
    short result = FALSE;
    VectorStr *vec = NULL;
    if (!data)
       goto cleanup;  //< Could become return false

    // ... Calculation which 'new's VectorStr

    result = TRUE;
cleanup:
    delete [] vec;
    return result;
}
```

In C ++ kann man RAII verwenden , um dieses Problem zu beheben:

```
struct VectorRAII final {
    VectorStr *data{nullptr};
    VectorRAII() = default;
    ~VectorRAII() {
        delete [] data;
    }
    VectorRAII(const VectorRAII &) = delete;
};

short calculate(VectorStr **data) {
    VectorRAII vec{};
    if (!data)
       return FALSE;  //< Could become return false

    // ... Calculation which 'new's VectorStr and stores it in vec.data

    return TRUE;
}
```

Ab diesem Zeitpunkt könnte der eigentliche Code weiter überarbeitet werden. Zum Beispiel durch Ersetzen des `VectorRAII` durch `std::unique_ptr` oder `std::vector` .

Refactoring-Techniken online lesen: https://riptutorial.com/de/cplusplus/topic/7600/refactoring-techniken

Kapitel 92: Reguläre Ausdrücke

Einführung

Reguläre Ausdrücke (manchmal auch Regex oder Regex genannt) sind eine Textsyntax, die die Muster darstellt, die in den bearbeiteten Strings abgeglichen werden können.

Reguläre Ausdrücke, die in c ++ 11 eingeführt wurden , unterstützen optional ein Rückgabefeld aus übereinstimmenden Zeichenfolgen oder eine andere Textsyntax, die definiert, wie übereinstimmende Muster in Zeichenfolgen ersetzt werden, für die ein Vorgang ausgeführt wird.

Syntax

- regex_match // Gibt zurück, ob die gesamte Zeichenfolge vom regulären Ausdruck abgeglichen wurde, wobei optional ein Übereinstimmungsobjekt erfasst wird
- regex_search // Gibt zurück, ob ein Teil der Zeichenfolge vom regulären Ausdruck abgeglichen wurde, wobei optional ein Übereinstimmungsobjekt erfasst wird
- regex_replace // Gibt die Eingabezeichenfolge zurück, wie sie von einem regulären Ausdruck geändert wurde, und zwar über einen Ersetzungsformat-String
- regex_token_iterator // Initialisiert mit einer von Iteratoren definierten Zeichenfolge, einer Liste von Erfassungsindizes, die durchlaufen werden sollen, und einer Regex. Dereferencing gibt die aktuell indizierte Übereinstimmung der Regex zurück. Durch Inkrementieren zum nächsten Capture-Index oder wenn aktuell im letzten Index, wird der Index zurückgesetzt und das nächste Vorkommen einer Regex-Übereinstimmung in der Zeichenfolge verhindert
- regex_iterator // Mit einer durch Iteratoren definierten Zeichenfolge und Regex initialisiert. Dereferencing gibt den Teil der Zeichenfolge zurück, auf den der gesamte reguläre Ausdruck passt. Beim Inkrementieren wird das nächste Vorkommen einer Regex-Übereinstimmung in der Zeichenfolge gefunden

Parameter

Unterschrift	Beschreibung
`bool regex_match(BidirectionalIterator first, BidirectionalIterator last, smatch& sm, const regex& re, regex_constraints::match_flag_type flags)`	**`BidirectionalIterator`** ist jedes Zeichen Iterator, Zuwachs und Dekrementoperatoren bietet **`smatch`** sein kann `cmatch` oder jede andere andere Variante von `match_results` , die die Art der nimmt `BidirectionalIterator` das `smatch` Argument weggelassen werden, wenn die Ergebnisse der Regex nicht **Returns** benötigt , ob `re` den gesamten Zeichenspiele Reihenfolge definiert als `first` und `last`
`bool regex_match(const string& str, smatch& sm, const regex re&,`	**`string`** kann entweder eine sein `const char*` oder

Unterschrift	Beschreibung
`regex_constraints::match_flag_type flags)`	eine L-Wert - `string` , *die Funktionen einen R-Wert zu übernehmen* `string` *explizit gelöscht* **`smatch`** kann `cmatch` oder jede andere andere Variante von `match_results` , die die Art von übernimmt `str` das `smatch` Argument weggelassen werden , wenn Die Ergebnisse des regulären Ausdrucks werden nicht benötigt. **Gibt zurück,** ob `re` mit der gesamten durch `str` definierten Zeichenfolge übereinstimmt

Examples

Grundlegende Beispiele für "regex_match" und "regex_search"

```
const auto input = "Some people, when confronted with a problem, think \"I know, I'll use
regular expressions.\""s;
smatch sm;

cout << input << endl;

// If input ends in a quotation that contains a word that begins with "reg" and another word
begining with "ex" then capture the preceeding portion of input
if (regex_match(input, sm, regex("(.*)\".*\\breg.*\\bex.*\"\\s*$"))) {
    const auto capture = sm[1].str();

    cout << '\t' << capture << endl; // Outputs: "\tSome people, when confronted with a
problem, think\n"

    // Search our capture for "a problem" or "# problems"
    if(regex_search(capture, sm, regex("(a|d+)\\s+problems?"))) {
        const auto count = sm[1] == "a"s ? 1 : stoi(sm[1]);

        cout << '\t' << count << (count > 1 ? " problems\n" : " problem\n"); // Outputs: "\t1
problem\n"
        cout << "Now they have " << count + 1 << " problems.\n"; // Ouputs: "Now they have 2
problems\n"
    }
}
```

Live-Beispiel

regex_replace Beispiel

Dieser Code berücksichtigt verschiedene Klammerstile und konvertiert sie in einen echten Klammerstil :

```
const auto input = "if (KnR)\n\tfoo();\nif (spaces) {\n    foo();\n}\nif
(allman)\n{\n\tfoo();\n}\nif (horstmann)\n{\tfoo();\n}\nif (pico)\n{\tfoo(); }\nif
(whitesmiths)\n\t{\n\tfoo();\n\t}\n"s;
```

```
cout << input << regex_replace(input, regex("(.+?)\\s*\\{?\\s*(.+?;)\\s*\\}?\\s*"), "$1
{\n\t$2\n}\n") << endl;
```

Live-Beispiel

regex_token_iterator Beispiel

Ein `std::regex_token_iterator` bietet ein enormes Werkzeug zum Extrahieren von Elementen einer Comma Separated Value-Datei . Abgesehen von den Vorteilen der Iteration kann dieser Iterator auch Kommas einfangen, wenn andere Methoden Probleme haben:

```
const auto input = "please split,this,csv, ,line,\\,\n"s;
const regex re{ "((?:[^\\\\,]|\\\\.)+)(?:,|$)" };
const vector<string> m_vecFields{ sregex_token_iterator(cbegin(input), cend(input), re, 1),
sregex_token_iterator() };

cout << input << endl;

copy(cbegin(m_vecFields), cend(m_vecFields), ostream_iterator<string>(cout, "\n"));
```

Live-Beispiel

Ein bemerkenswertes Problem bei Regex-Iteratoren ist, dass das `regex` Argument ein L-Wert sein muss. Ein R-Wert funktioniert nicht .

regex_iterator Beispiel

Wenn die Verarbeitung von Captures iterativ durchgeführt werden `regex_iterator` ist ein `regex_iterator` eine gute Wahl. Wenn ein `regex_iterator` ein `match_result` . Dies ist ideal für konditionale Captures oder Captures mit wechselseitiger Abhängigkeit. Nehmen wir an, wir wollen etwas C ++ - Code tokenisieren. Gegeben:

```
enum TOKENS {
    NUMBER,
    ADDITION,
    SUBTRACTION,
    MULTIPLICATION,
    DIVISION,
    EQUALITY,
    OPEN_PARENTHESIS,
    CLOSE_PARENTHESIS
};
```

Wir können diese Zeichenkette mit einem Token versehen: `const auto input = "42/2 + -8\t=\n(2 + 2) * 2 * 2 -3"s` mit einem `regex_iterator` wie `regex_iterator` :

```
vector<TOKENS> tokens;
const regex re{ "\\s*(\\(?)\\s*(-?\\s*\\d+)\\s*(\\)?)\\s*(?:(\\+)|(-)|(\\*)|(/)|(=))" };

for_each(sregex_iterator(cbegin(input), cend(input), re), sregex_iterator(), [&](const auto&
i) {
    if(i[1].length() > 0) {
```

```
        tokens.push_back(OPEN_PARENTHESIS);
    }

    tokens.push_back(i[2].str().front() == '-' ? NEGATIVE_NUMBER : NON_NEGATIVE_NUMBER);

    if(i[3].length() > 0) {
        tokens.push_back(CLOSE_PARENTHESIS);
    }

    auto it = next(cbegin(i), 4);

    for(int result = ADDITION; it != cend(i); ++result, ++it) {
        if (it->length() > 0U) {
            tokens.push_back(static_cast<TOKENS>(result));
            break;
        }
    }
});

match_results<string::const_reverse_iterator> sm;

if(regex_search(crbegin(input), crend(input), sm, regex{ tokens.back() == SUBTRACTION ?
"^\\s*\\d+\\s*-\\s*(-?)" : "^\\s*\\d+\\s*(-?)" })) {
    tokens.push_back(sm[1].length() == 0 ? NON_NEGATIVE_NUMBER : NEGATIVE_NUMBER);
}
```

Live-Beispiel

Ein bemerkenswertes Argument bei Regex-Iteratoren ist, dass das `regex` Argument ein L-Wert sein muss. Ein R-Wert funktioniert nicht: Visual Studio regex_iterator Fehler?

Einen String aufteilen

```
std::vector<std::string> split(const std::string &str, std::string regex)
{
    std::regex r{ regex };
    std::sregex_token_iterator start{ str.begin(), str.end(), r, -1 }, end;
    return std::vector<std::string>(start, end);
}
```

```
split("Some  string\t with whitespace ", "\\s+"); // "Some", "string", "with", "whitespace"
```

Quantifizierer

Nehmen wir an, wir haben eine `const string input` als zu überprüfende Telefonnummer. Wir könnten mit einer numerischen Eingabe mit einem **Null- oder mehr-Quantifizierer beginnen** : `regex_match(input, regex("\\d*"))` oder einem **oder mehreren Quantifizierer** : `regex_match(input, regex("\\d+"))` Beide sind wirklich unzureichend, wenn die `input` eine ungültige numerische Zeichenfolge enthält: "123". Verwenden Sie einen **Quantifizierer n oder mehr,** um sicherzustellen, dass wir mindestens 7 Ziffern erhalten:

```
regex_match(input, regex("\\d{7,}"))
```

Dies garantiert, dass wir mindestens eine Telefonnummer mit Ziffern erhalten, aber die `input` könnte auch eine numerische Zeichenfolge enthalten, die zu lang ist: "123456789012". Also mit einem **Quantifizierer zwischen n und m,** so dass die `input` mindestens 7 Ziffern, aber nicht mehr als 11 hat:

```
regex_match(input, regex("\\d{7,11}"));
```

Dies bringt uns näher, aber illegale numerische Zeichenfolgen, die im Bereich von [7, 11] liegen, werden immer noch akzeptiert, wie: "123456789". Lassen Sie uns den Ländercode mit einem **Lazy-Quantifizierer** optional machen:

```
regex_match(input, regex("\\d?\\d{7,10}"))
```

Es ist wichtig zu wissen, dass der **faule Quantifizierer** *so wenig Zeichen wie möglich enthält. Die* Übereinstimmung *des Zeichens* ist daher nur *möglich* , wenn bereits 10 Zeichen mit `\d{7,10}` übereinstimmen. (Um das erste Zeichen gierig anzupassen, müssten wir Folgendes tun: `\d{0,1}` .) Der **faule Quantifizierer** kann an jeden anderen Quantifizierer angehängt werden.

Wie würden wir die Vorwahl optional machen *und* eine Landesvorwahl nur akzeptieren, wenn die Vorwahl vorhanden ist?

```
regex_match(input, regex("(?:\\d{3,4})?\\d{7}"))
```

In diesem letzten regulären Ausdruck, der `\d{7}` *erfordert* 7 Stellen. Vor diesen 7 Ziffern stehen wahlweise 3 oder 4 Ziffern.

Beachten Sie, dass der **faule Quantifizierer** nicht angehängt wurde: ~~`\d{3,4}?\d{7}`~~ , der `\d{3,4}?` hätte entweder 3 oder 4 Zeichen gefunden und würde 3 bevorzugen. Stattdessen machen wir die Gruppe, die keine Capture erfasst, höchstens einmal. Ursache einer Nichtübereinstimmung, wenn die `input` keine Vorwahl wie "1234567" enthielt.

Zum Abschluss des Quantifikator-Themas möchte ich den anderen anhängigen Quantifizierer erwähnen, den Sie verwenden können, den **Possessiv-Quantifizierer** . *Entweder* der **Lazy Quantifier** oder der **Possessive Quantifier** kann an einen beliebigen Quantifier angehängt werden. Die einzige Funktion des **Possessiv-Quantifizierers** besteht darin, die Regex-Engine zu unterstützen, indem sie sie dazu auffordert, diese Charaktere gierig zu nehmen *und sie niemals aufzugeben, selbst wenn dadurch der Regex fehlschlägt* . Dies macht zum Beispiel nicht viel Sinn: `regex_match(input, regex("\\d{3,4}+\\d{7}))` Da eine `input` wie:" 1234567890 "nicht als `\d{3,4}+` übereinstimmen würde `\d{3,4}+` stimmt immer mit 4 Zeichen überein, selbst wenn die Übereinstimmung mit 3 den Regex erfolgreich gemacht hätte.
Der **Possessiv-Quantifizierer** wird am besten verwendet, *wenn das quantifizierte Token die Anzahl anpassbarer Zeichen begrenzt* . Zum Beispiel:

```
regex_match(input, regex("(?:.*\\d{3,4}+){3}"))
```

Kann verwendet werden, wenn die `input` eine der folgenden `input` enthält:

123 456 7890
123-456-7890
(123)456-7890
(123) 456 - 7890

Wenn dieser Regex wirklich glänzt, dann ist die `input` eine *unzulässige* Eingabe:

12345 - 67890

Ohne den **Possessiv-Quantifizierer muss** die Regex-Engine zurückgehen und *jede Kombination von* `.*` *Und entweder 3 oder 4 Zeichen* testen, *um* zu sehen, ob sie eine übereinstimmende Kombination findet. Mit dem **besitzergreifend quantifier** wo die regulären Ausdruck beginnt die 2. **besitzergreifend quantifier** aufgehört hat , die ‚0', und der Motor versucht der regex einzustellen . `.*` Ermöglichen `\d{3,4}` zu entsprechen; wenn es nicht die Regex nur fehlschlägt, wird keine Rückverfolgung zu sehen getan , wenn früher `.*` Einstellung ein Spiel erlaubt hätte.

Anker

C ++ bietet nur 4 Anker:

- `^` was den Start der Zeichenkette bestätigt
- `$` die das Ende der Zeichenfolge bestätigt
- `\b` das ein `\W` Zeichen oder den Anfang oder das Ende der Zeichenfolge bestätigt
- `\B` das ein `\w` -Zeichen bestätigt

Nehmen wir zum Beispiel an, wir möchten eine Zahl *mit* ihrem Vorzeichen erfassen:

```
auto input = "+1--12*123/+1234"s;
smatch sm;

if(regex_search(input, sm, regex{ "(?:^|\\b\\W)([+-]?\\d+)" })) {

    do {
        cout << sm[1] << endl;
        input = sm.suffix().str();
    } while(regex_search(input, sm, regex{ "(?:^\\W|\\b\\W)([+-]?\\d+)" }));
}
```

Live-Beispiel

Ein wichtiger Hinweis hierbei ist, dass der Anker keine Zeichen verbraucht.

Reguläre Ausdrücke online lesen: https://riptutorial.com/de/cplusplus/topic/1681/regulare-ausdrucke

Kapitel 93: Rekursion in C ++

Examples

Verwenden der Schwanzrekursion und Fibonnaci-Rekursion, um die Fibonnaci-Sequenz zu lösen

Der einfachste und naheliegendste Weg, Rekursion zu verwenden, um den N-ten Term der Fibonnaci-Sequenz zu erhalten, ist dies

```
int get_term_fib(int n)
{
  if (n == 0)
    return 0;
  if (n == 1)
    return 1;
  return get_term_fib(n - 1) + get_term_fib(n - 2);
}
```

Dieser Algorithmus skaliert jedoch nicht für höhere Ausdrücke: Für immer größere `n` nimmt die Anzahl der Funktionsaufrufe, die Sie machen müssen, exponentiell zu. Dies kann durch eine einfache Schwanzrekursion ersetzt werden.

```
int get_term_fib(int n, int prev = 0, int curr = 1)
{
  if (n == 0)
    return prev;
  if (n == 1)
    return curr;
  return get_term_fib(n - 1, curr, prev + curr);
}
```

Jeder Aufruf der Funktion berechnet jetzt sofort den nächsten Term in der Fibonnaci-Sequenz, sodass die Anzahl der Funktionsaufrufe linear mit `n` skaliert.

Rekursion mit Memoisierung

Rekursive Funktionen können recht teuer werden. Wenn es sich um reine Funktionen handelt (Funktionen, die beim Aufruf mit denselben Argumenten immer denselben Wert zurückgeben und die weder vom externen Zustand abhängen noch diesen ändern), können sie auf Kosten des Speichers durch Speichern der bereits berechneten Werte erheblich schneller gemacht werden.

Das Folgende ist eine Implementierung der Fibonacci-Sequenz mit Memoisierung:

```
#include <map>

int fibonacci(int n)
{
  static std::map<int, int> values;
```

```
  if (n==0 || n==1)
    return n;
  std::map<int,int>::iterator iter = values.find(n);
  if (iter == values.end())
  {
    return values[n] = fibonacci(n-1) + fibonacci(n-2);
  }
  else
  {
    return iter->second;
  }
}
```

Beachten Sie, dass diese Funktion trotz der einfachen Rekursionsformel beim ersten Aufruf $ O (n) $ ist. Bei nachfolgenden Aufrufen mit dem gleichen Wert ist es natürlich $ O (1) $.

Beachten Sie jedoch, dass diese Implementierung nicht wiedereintrittsfähig ist. Außerdem können gespeicherte Werte nicht gelöscht werden. Eine alternative Implementierung wäre, die Map als zusätzliches Argument übergeben zu lassen:

```
#include <map>

int fibonacci(int n, std::map<int, int> values)
{
  if (n==0 || n==1)
    return n;
  std::map<int,int>::iterator iter = values.find(n);
  if (iter == values.end())
  {
    return values[n] = fibonacci(n-1) + fibonacci(n-2);
  }
  else
  {
    return iter->second;
  }
}
```

Bei dieser Version muss der Anrufer die Karte mit den gespeicherten Werten verwalten. Dies hat den Vorteil, dass die Funktion jetzt wiedereintrittsfähig ist und der Aufrufer nicht mehr benötigte Werte entfernen kann, um Speicherplatz zu sparen. Es hat den Nachteil, dass es die Einkapselung bricht; Der Aufrufer kann die Ausgabe ändern, indem er die Karte mit falschen Werten auffüllt.

Rekursion in C ++ online lesen: https://riptutorial.com/de/cplusplus/topic/5693/rekursion-in-c-plusplus

Kapitel 94: Rekursiver Mutex

Examples

std :: recursive_mutex

Rekursiver Mutex ermöglicht, dass derselbe Thread eine Ressource rekursiv sperrt - bis zu einem nicht angegebenen Grenzwert.

Dafür gibt es nur sehr wenige echte Begründungen. Bestimmte komplexe Implementierungen müssen möglicherweise eine überladene Kopie einer Funktion aufrufen, ohne die Sperre aufzuheben.

```
    std::atomic_int temp{0};
    std::recursive_mutex _mutex;

    //launch_deferred launches asynchronous tasks on the same thread id

    auto future1 = std::async(
                std::launch::deferred,
                [&]()
                {
                    std::cout << std::this_thread::get_id() << std::endl;

                    std::this_thread::sleep_for(std::chrono::seconds(3));
                    std::unique_lock<std::recursive_mutex> lock( _mutex);
                    temp=0;

                });

    auto future2 = std::async(
                std::launch::deferred,
                [&]()
                {
                    std::cout << std::this_thread::get_id() << std::endl;
                    while ( true )
                    {
                        std::this_thread::sleep_for(std::chrono::milliseconds(1));
                        std::unique_lock<std::recursive_mutex> lock( _mutex,
std::try_to_lock);
                        if ( temp < INT_MAX )
                             temp++;

                        cout << temp << endl;

                    }
                });
    future1.get();
    future2.get();
```

Rekursiver Mutex online lesen: https://riptutorial.com/de/cplusplus/topic/9929/rekursiver-mutex

Kapitel 95: Resourcenmanagement

Einführung

In C und C ++ ist das Ressourcenmanagement eine der schwierigsten Aufgaben. Zum Glück haben wir in C ++ viele Möglichkeiten, Ressourcenmanagement in unseren Programmen zu gestalten. Dieser Artikel soll einige der Redewendungen und Methoden erläutern, die zur Verwaltung der zugewiesenen Ressourcen verwendet werden.

Examples

Ressourcenakquisition ist Initialisierung

Die Ressourcenerfassung ist die Initialisierung (RAII) ist ein allgemeiner Ausdruck in der Ressourcenverwaltung. Im Fall von dynamischem Speicher verwendet es intelligente Zeiger , um das Ressourcenmanagement durchzuführen. Bei Verwendung von RAII wird eine erworbene Ressource sofort einem intelligenten Zeiger oder einem gleichwertigen Ressourcenmanager zugeordnet. Auf die Ressource kann nur über diesen Manager zugegriffen werden, sodass der Manager verschiedene Vorgänge verfolgen kann. Beispiel: `std::auto_ptr` die entsprechende Ressource automatisch frei, wenn sie außerhalb des Gültigkeitsbereichs liegt oder anderweitig gelöscht wird.

```
#include <memory>
#include <iostream>
using namespace std;

int main() {
    {
        auto_ptr ap(new int(5)); // dynamic memory is the resource
        cout << *ap << endl; // prints 5
    } // auto_ptr is destroyed, its resource is automatically freed
}
```

C ++ 11

Das Hauptproblem von `std::auto_ptr` ist, dass es nicht kopiert werden kann, ohne den Besitzer zu übertragen:

```
#include <memory>
#include <iostream>
using namespace std;

int main() {
    auto_ptr ap1(new int(5));
    cout << *ap1 << endl; // prints 5
    auto_ptr ap2(ap1); // copy ap2 from ap1; ownership now transfers to ap2
    cout << *ap2 << endl; // prints 5
    cout << ap1 == nullptr << endl; // prints 1; ap1 has lost ownership of resource
}
```

Aufgrund dieser seltsamen Kopiersemantik kann `std::auto_ptr` unter anderem nicht in Containern verwendet werden. Der Grund ist, dass das Löschen des Speichers zweimal verhindert wird: Wenn zwei `auto_ptrs` mit der gleichen Ressource vorhanden sind, versuchen beide, sie zu löschen, wenn sie zerstört werden. Das Freigeben einer bereits freigegebenen Ressource kann im Allgemeinen zu Problemen führen. Daher ist es wichtig, dies zu verhindern. `std::shared_ptr` hat jedoch eine Methode, um dies zu vermeiden, während beim Kopieren der Besitz nicht übertragen wird:

```
#include <memory>
#include <iostream>
using namespace std;

int main() {
    shared_ptr sp2;
    {
        shared_ptr sp1(new int(5)); // give ownership to sp1
        cout << *sp1 << endl; // prints 5
        sp2 = sp1; // copy sp2 from sp1; both have ownership of resource
        cout << *sp1 << endl; // prints 5
        cout << *sp2 << endl; // prints 5
    } // sp1 goes out of scope and is destroyed; sp2 has sole ownership of resource
    cout << *sp2 << endl;
} // sp2 goes out of scope; nothing has ownership, so resource is freed
```

Mutexe & Fadensicherheit

Probleme können auftreten, wenn mehrere Threads versuchen, auf eine Ressource zuzugreifen. Angenommen, wir haben einen Thread, der einer Variablen einen hinzufügt. Dazu liest man zuerst die Variable, fügt eine hinzu und speichert sie dann zurück. Angenommen, wir initialisieren diese Variable auf 1 und erstellen dann zwei Instanzen dieses Threads. Nachdem beide Threads abgeschlossen sind, schlägt die Intuition vor, dass diese Variable den Wert 3 haben sollte. Die folgende Tabelle veranschaulicht jedoch, was möglicherweise schief geht:

	Faden 1	Faden 2
Zeitschritt 1	Lese 1 aus Variable	
Zeitschritt 2		Lese 1 aus Variable
Zeitschritt 3	Addiere 1 plus 1, um 2 zu erhalten	
Zeitschritt 4		Addiere 1 plus 1, um 2 zu erhalten
Zeitschritt 5	Speichern Sie 2 in Variable	
Zeitschritt 6		Speichern Sie 2 in Variable

Wie Sie sehen, befindet sich am Ende der Operation die Variable 2 anstelle von 3. Der Grund ist, dass der Thread 2 die Variable gelesen hat, bevor der Thread 1 seine Aktualisierung durchgeführt hat. Die Lösung? Mutexe

Ein Mutex (Kunstwort aus **mut** ual **ex** schluss) ist ein Objekt Ressourcenmanagement entwickelt , um diese Art von Problem zu lösen. Wenn ein Thread auf eine Ressource zugreifen möchte, "erhält" er den Mutex der Ressource. Sobald der Zugriff auf die Ressource abgeschlossen ist, "gibt" der Thread den Mutex frei. Während der Mutex erworben wird, kehren alle Aufrufe zum Erwerb des Mutex nicht zurück, bis der Mutex freigegeben wird. Um dies besser zu verstehen, stellen Sie sich einen Mutex als Warteschlange im Supermarkt vor: Die Fäden gehen in die Reihe, indem sie versuchen, den Mutex zu erhalten, und warten, bis die vor ihnen liegenden Fäden fertiggestellt sind, dann die Ressource nutzen und dann aussteigen Linie durch Freigabe des Mutex. Wenn alle versuchen, sofort auf die Ressource zuzugreifen, gäbe es ein komplettes Pandemium.

C ++ 11

`std::mutex` ist die Implementierung eines Mutex in C ++ 11.

```
#include <thread>
#include <mutex>
#include <iostream>
using namespace std;

void add_1(int& i, const mutex& m) { // function to be run in thread
    m.lock();
    i += 1;
    m.unlock();
}

int main() {
    int var = 1;
    mutex m;

    cout << var << endl; // prints 1

    thread t1(add_1, var, m); // create thread with arguments
    thread t2(add_1, var, m); // create another thread
    t1.join(); t2.join(); // wait for both threads to finish

    cout << var << endl; // prints 3
}
```

Resourcenmanagement online lesen:
https://riptutorial.com/de/cplusplus/topic/8336/resourcenmanagement

Kapitel 96: RTTI: Informationen zum Laufzeit-Typ

Examples

Name eines Typs

Sie können den implementierungsdefinierten Namen eines Typs in Runtime abrufen, indem Sie die Member-Funktion `.name()` des von `typeid` `std::type_info` Objekts `std::type_info` `typeid` .

```
#include <iostream>
#include <typeinfo>

int main()
{
    int speed = 110;

    std::cout << typeid(speed).name() << '\n';
}
```

Ausgabe (implementierungsdefiniert):

```
int
```

dynamischer_cast

Verwenden Sie `dynamic_cast<>()` als Funktion, die Ihnen hilft, eine Vererbungshierarchie (Hauptbeschreibung) `dynamic_cast<>()` .

Wenn Sie für einige abgeleitete Klassen `B` und `C` einige nicht polymorphe Arbeiten ausführen müssen, aber die Basisklasse `class A` , schreiben Sie folgendermaßen

```
class A { public: virtual ~A(){} };

class B: public A
{ public: void work4B(){} };

class C: public A
{ public: void work4C(){} };

void non_polymorphic_work(A* ap)
{
  if (B* bp =dynamic_cast<B*>(ap))
    bp->work4B();
  if (C* cp =dynamic_cast<C*>(ap))
    cp->work4C();
}
```

Das typid-Schlüsselwort

Das Schlüsselwort `typeid` ist ein `typeid` Operator, der Informationen zum Laufzeittyp über seinen Operanden liefert, wenn der Typ des Operanden ein polymorpher Klassentyp ist. Es gibt einen Wert vom Typ `const std::type_info` . Lebenslauf-Qualifikation auf oberster Ebene wird ignoriert.

```
struct Base {
    virtual ~Base() = default;
};
struct Derived : Base {};
Base* b = new Derived;
assert(typeid(*b) == typeid(Derived{})); // OK
```

`typeid` kann auch direkt auf einen Typ angewendet werden. In diesem Fall werden die ersten Verweise auf oberster Ebene entfernt, und die cv-Qualifizierung auf oberster Ebene wird ignoriert. Das obige Beispiel hätte also mit `typeid(Derived)` anstelle von `typeid(Derived{})` :

```
assert(typeid(*b) == typeid(Derived{})); // OK
```

Wenn `typeid` auf einen Ausdruck angewendet wird, der *nicht* vom polymorphen Klassentyp ist, wird der Operand nicht ausgewertet, und die Typinfo wird für den statischen Typ zurückgegeben.

```
struct Base {
    // note: no virtual destructor
};
struct Derived : Base {};
Derived d;
Base& b = d;
assert(typeid(b) == typeid(Base)); // not Derived
assert(typeid(std::declval<Base>()) == typeid(Base)); // OK because unevaluated
```

Wann welcher Cast in C ++ verwendet wird

Verwenden Sie **dynamic_cast** zum Konvertieren von Zeigern / Referenzen innerhalb einer Vererbungshierarchie.

Verwenden Sie **static_cast** für normale Typkonvertierungen.

Verwenden Sie **reinterpret_cast,** um Bitmuster auf niedriger Ebene neu zu interpretieren. Mit äußerster Vorsicht verwenden.

Verwenden Sie **const_cast,** um const / volatile zu **verwerfen** . Vermeiden Sie dies, wenn Sie nicht mit einer const-wrong API arbeiten.

RTTI: Informationen zum Laufzeit-Typ online lesen:
https://riptutorial.com/de/cplusplus/topic/3129/rtti--informationen-zum-laufzeit-typ

Kapitel 97: Rückgabetyp Kovarianz

Bemerkungen

Bei der Kovarianz eines Parameters oder eines Rückgabewerts für eine virtuelle Elementfunktion `m` wird der Typ `T` in einer abgeleiteten Klasse von `m` überschrieben. Der Typ `T` variiert dann (*Varianz*) in der Spezifität auf dieselbe Weise (*co*) wie die Klassen, die `m` bereitstellen. C ++ bietet Sprachunterstützung für kovariante *Rückgabetypen, bei* denen es sich um Rohzeiger oder Rohreferenzen handelt. Die Kovarianz bezieht sich auf den Pointee- oder Referenztyp.

Die C ++ - Unterstützung ist auf Rückgabetypen beschränkt, da Funktionsrückgabewerte die einzigen reinen **Out-Argumente** in C ++ sind und Kovarianz nur für ein reines Out-Argument typsicher ist. Andernfalls könnte der aufrufende Code ein Objekt eines weniger spezifischen Typs liefern, als der empfangende Code erwartet. Die MIT-Professorin Barbara Liskov untersuchte diese und verwandte Sicherheitsprobleme des Typs "Abweichungstyp" und ist jetzt als Liskov-Substitutionsprinzip (**LSP) bekannt** .

Die Kovarianz-Unterstützung hilft im Wesentlichen, Downcasting und dynamische Typüberprüfung zu vermeiden.

Da Smart-Pointer vom Klassentyp sind, kann die integrierte Unterstützung für Kovarianz nicht direkt für Smart-Pointer-Ergebnisse verwendet werden. Man kann jedoch *scheinbar kovariante* nicht- `virtual` Smart-Pointer-Wrapper-Funktionen für eine kovariante `virtual` Funktion definieren, die rohe Pointer erzeugt.

Examples

1. Basisbeispiel ohne kovariante Renditen zeigt, warum sie wünschenswert sind

```
// 1. Base example not using language support for covariance, dynamic type checking.

class Top
{
public:
    virtual Top* clone() const = 0;
    virtual ~Top() = default;       // Necessary for `delete` via Top*.
};

class D : public Top
{

public:
    Top* clone() const override
    { return new D( *this ); }
};

class DD : public D
{
```

```
private:
    int answer_ = 42;

public:
    int answer() const
    { return answer_;}

    Top* clone() const override
    { return new DD( *this ); }
};

#include <assert.h>
#include <iostream>
#include <typeinfo>
using namespace std;

int main()
{
    cout << boolalpha;

    DD* p1 = new DD();
    Top* p2 = p1->clone();
    bool const  correct_dynamic_type = (typeid( *p2 ) == typeid( DD ));
    cout << correct_dynamic_type << endl;               // "true"

    assert( correct_dynamic_type ); // This is essentially dynamic type checking. :(
    auto p2_most_derived = static_cast<DD*>( p2 );
    cout << p2_most_derived->answer() << endl;          // "42"
    delete p2;
    delete p1;
}
```

2. Covariante Ergebnisversion des Basisbeispiels, statische Typüberprüfung.

```
// 2. Covariant result version of the base example, static type checking.

class Top
{
public:
    virtual Top* clone() const = 0;
    virtual ~Top() = default;       // Necessary for `delete` via Top*.
};

class D : public Top
{
public:
    D* /* ← Covariant return */ clone() const override
    { return new D( *this ); }
};

class DD : public D
{
private:
    int answer_ = 42;

public:
    int answer() const
    { return answer_;}
```

```
    DD* /* ← Covariant return */ clone() const override
    { return new DD( *this ); }
};

#include <iostream>
using namespace std;

int main()
{
    DD* p1 = new DD();
    DD* p2 = p1->clone();
    // Correct dynamic type DD for *p2 is guaranteed by the static type checking.

    cout << p2->answer() << endl;          // "42"
    delete p2;
    delete p1;
}
```

3. Covariant Smart Pointer-Ergebnis (automatisierte Bereinigung).

```
// 3. Covariant smart pointer result (automated cleanup).

#include <memory>
using std::unique_ptr;

template< class Type >
auto up( Type* p ) { return unique_ptr<Type>( p ); }

class Top
{
private:
    virtual Top* virtual_clone() const = 0;

public:
    unique_ptr<Top> clone() const
    { return up( virtual_clone() ); }

    virtual ~Top() = default;       // Necessary for `delete` via Top*.
};

class D : public Top
{
private:
    D* /* ← Covariant return */ virtual_clone() const override
    { return new D( *this ); }

public:
    unique_ptr<D> /* ← Apparent covariant return */ clone() const
    { return up( virtual_clone() ); }
};

class DD : public D
{
private:
    int answer_ = 42;

    DD* /* ← Covariant return */ virtual_clone() const override
    { return new DD( *this ); }
```

```
public:
    int answer() const
    { return answer_;}

    unique_ptr<DD> /* ← Apparent covariant return */ clone() const
    { return up( virtual_clone() ); }
};

#include <iostream>
using namespace std;

int main()
{
    auto  p1 = unique_ptr<DD>(new DD());
    auto  p2 = p1->clone();
    // Correct dynamic type DD for *p2 is guaranteed by the static type checking.

    cout << p2->answer() << endl;          // "42"
    // Cleanup is automated via unique_ptr.
 }
```

Rückgabetyp Kovarianz online lesen: https://riptutorial.com/de/cplusplus/topic/5411/ruckgabetyp-kovarianz

Kapitel 98: Schleifen

Einführung

Eine Schleifenanweisung führt eine Gruppe von Anweisungen wiederholt aus, bis eine Bedingung erfüllt ist. Es gibt drei Arten von primitiven Schleifen in C ++: für, while und do ... while.

Syntax

- while (*Bedingung*) *Anweisung* ;
- do- *Anweisung* while (*Ausdruck*);
- for (*for-init-Anweisung* ; *Bedingung* ; *Ausdruck*) *Anweisung* ;
- for (*für Range-Deklaration* : *für Range-Initialisierer*) *Anweisung* ;
- brechen ;
- fortsetzen ;

Bemerkungen

`algorithm` sind im Allgemeinen handgeschriebenen Schleifen vorzuziehen.

Wenn Sie etwas möchten, das ein Algorithmus bereits tut (oder etwas sehr ähnliches), ist der Algorithmusaufruf klarer, häufig effizienter und weniger fehleranfällig.

Wenn Sie eine Schleife benötigen, die recht einfache Funktionen ausführt (wenn Sie jedoch einen verwirrenden Knoten aus Bindern und Adaptern benötigen, wenn Sie einen Algorithmus verwenden), schreiben Sie einfach die Schleife.

Examples

Bereichsabhängig für

C ++ 11

`for` Schleifen können verwendet werden, um die Elemente eines Iterator-basierten Bereichs zu durchlaufen, ohne einen numerischen Index zu verwenden oder direkt auf die Iteratoren zuzugreifen:

```
vector<float> v = {0.4f, 12.5f, 16.234f};

for(auto val: v)
{
    std::cout << val << " ";
}

std::cout << std::endl;
```

Dadurch wird jedes Element in `v` durchlaufen, wobei `val` den Wert des aktuellen Elements erhält. Die folgende Aussage:

```
for (for-range-declaration : for-range-initializer ) statement
```

ist äquivalent zu:

```
{
    auto&& __range = for-range-initializer;
    auto __begin = begin-expr, __end = end-expr;
    for (; __begin != __end; ++__begin) {
        for-range-declaration = *__begin;
        statement
    }
}
```

C ++ 17

```
{
    auto&& __range = for-range-initializer;
    auto __begin = begin-expr;
    auto __end = end-expr; // end is allowed to be a different type than begin in C++17
    for (; __begin != __end; ++__begin) {
        for-range-declaration = *__begin;
        statement
    }
}
```

Diese Änderung wurde für die geplante Unterstützung von Ranges TS in C ++ 20 eingeführt.

In diesem Fall entspricht unsere Schleife:

```
{
    auto&& __range = v;
    auto __begin = v.begin(), __end = v.end();
    for (; __begin != __end; ++__begin) {
        auto val = *__begin;
        std::cout << val << " ";
    }
}
```

Beachten Sie, dass `auto val` einen `auto val` deklariert, bei dem es sich um eine Kopie eines im Bereich gespeicherten Werts handelt (wir initialisieren ihn vom Iterator aus). Wenn die in diesem Bereich gespeicherten Werte teuer zu kopieren sind, möchten Sie möglicherweise `const auto &val` . Sie müssen auch nicht `auto` . Sie können einen geeigneten Typnamen verwenden, sofern dieser implizit aus dem Werttyp des Bereichs konvertierbar ist.

Wenn Sie Zugriff auf den Iterator benötigen, kann Range-based für Sie nicht helfen (zumindest nicht ohne Anstrengung).

Wenn Sie darauf verweisen möchten, können Sie dies tun:

```
vector<float> v = {0.4f, 12.5f, 16.234f};
```

```
for(float &val: v)
{
    std::cout << val << " ";
}
```

Sie könnten die `const` Referenz durchlaufen, wenn Sie über einen `const` Container verfügen:

```
const vector<float> v = {0.4f, 12.5f, 16.234f};

for(const float &val: v)
{
    std::cout << val << " ";
}
```

Weiterleitungsreferenzen werden verwendet, wenn der Sequenz-Iterator ein Proxy-Objekt zurückgibt und Sie dieses Objekt auf eine nicht `const` Weise `const` . Hinweis: Die Leser Ihres Codes werden höchstwahrscheinlich verwirrt.

```
vector<bool> v(10);

for(auto&& val: v)
{
    val = true;
}
```

Der „Bereich“ type vorgesehen bereichsbasierte `for` eine der folgenden sein kann:

- Spracharrays:

  ```
  float arr[] = {0.4f, 12.5f, 16.234f};

  for(auto val: arr)
  {
      std::cout << val << " ";
  }
  ```

  Beachten Sie, dass das Zuordnen eines dynamischen Arrays nicht zählt:

  ```
  float *arr = new float[3]{0.4f, 12.5f, 16.234f};

  for(auto val: arr) //Compile error.
  {
      std::cout << val << " ";
  }
  ```

- Jeder Typ, der die Member-Funktionen `begin()` und `end()` , die Iteratoren an die Elemente des Typs zurückgeben. Die Standard-Bibliothekscontainer sind qualifiziert, aber auch benutzerdefinierte Typen können verwendet werden:

  ```
  struct Rng
  {
      float arr[3];
  ```

```
    // pointers are iterators
    const float* begin() const {return &arr[0];}
    const float* end() const   {return &arr[3];}
    float* begin() {return &arr[0];}
    float* end()   {return &arr[3];}
};

int main()
{
    Rng rng = {{0.4f, 12.5f, 16.234f}};

    for(auto val: rng)
    {
        std::cout << val << " ";
    }
}
```

- Jeder Typ, der über keine `begin(type)` und `end(type)` -Funktionen verfügt, die über eine artenabhängige Suche nach `type` . Dies ist nützlich, wenn Sie einen Bereichstyp erstellen möchten, ohne den Klassentyp selbst ändern zu müssen:

```
namespace Mine
{
    struct Rng {float arr[3];};

    // pointers are iterators
    const float* begin(const Rng &rng) {return &rng.arr[0];}
    const float* end(const Rng &rng) {return &rng.arr[3];}
    float* begin(Rng &rng) {return &rng.arr[0];}
    float* end(Rng &rng) {return &rng.arr[3];}
}

int main()
{
    Mine::Rng rng = {{0.4f, 12.5f, 16.234f}};

    for(auto val: rng)
    {
        std::cout << val << " ";
    }
}
```

Für Schleife

A `for` Schleife führt Anweisungen in dem `loop body` , während die `condition` wahr ist. Vor der Schleife wird die `initialization statement` genau einmal ausgeführt. Nach jedem Zyklus wird der `iteration execution` ausgeführt.

Eine `for` Schleife ist wie folgt definiert:

```
for (/*initialization statement*/; /*condition*/; /*iteration execution*/)
{
    // body of the loop
}
```

Erklärung der Platzhalteraussagen:

- `initialization statement` : Diese Anweisung wird nur einmal am Anfang der `for` Schleife ausgeführt. Sie können eine Deklaration mehrerer Variablen eines Typs eingeben, z. B. `int i = 0, a = 2, b = 3` . Diese Variablen sind nur im Geltungsbereich der Schleife gültig. Variablen, die vor der Schleife mit demselben Namen definiert wurden, werden während der Ausführung der Schleife ausgeblendet.
- `condition` : Diese Aussage ausgewertet wird vor jedem *Schleifenkörper* Ausführung, und bricht die Schleife , wenn es wertet `false` .
- `iteration execution` : Diese Anweisung wird nach dem Loop- *Body* vor der nächsten *Bedingungsbewertung ausgeführt* , es sei denn, die `for` Schleife wird im *Body* abgebrochen (durch `break` , `goto` , `return` oder eine geworfene Ausnahme). Sie können im `iteration execution` mehrere Anweisungen eingeben, z. B. `a++, b+=10, c=b+a` .

Das grobe Äquivalent einer `for` Schleife, die als `while` Schleife neu geschrieben wurde, lautet:

```
/*initialization*/
while (/*condition*/)
{
    // body of the loop; using 'continue' will skip to increment part below
    /*iteration execution*/
}
```

Der häufigste Fall für die Verwendung einer `for` Schleife ist das Ausführen von Anweisungen zu einer bestimmten Anzahl von Malen. Betrachten Sie zum Beispiel Folgendes:

```
for(int i = 0; i < 10; i++) {
    std::cout << i << std::endl;
}
```

Eine gültige Schleife ist auch:

```
for(int a = 0, b = 10, c = 20; (a+b+c < 100); c--, b++, a+=c) {
    std::cout << a << " " << b << " " << c << std::endl;
}
```

Ein Beispiel für das Ausblenden deklarierter Variablen vor einer Schleife ist:

```
int i = 99; //i = 99
for(int i = 0; i < 10; i++) { //we declare a new variable i
    //some operations, the value of i ranges from 0 to 9 during loop execution
}
//after the loop is executed, we can access i with value of 99
```

Wenn Sie jedoch die bereits deklarierte Variable verwenden und nicht ausblenden möchten, lassen Sie den Deklarationsteil aus:

```
int i = 99; //i = 99
for(i = 0; i < 10; i++) { //we are using already declared variable i
    //some operations, the value of i ranges from 0 to 9 during loop execution
}
```

```
//after the loop is executed, we can access i with value of 10
```

Anmerkungen:

- Die Initialisierungs- und Inkrementierungsanweisungen können Operationen ausführen, die nicht mit der Bedingungsanweisung zusammenhängen, oder gar nichts - falls Sie dies wünschen. Aus Gründen der Lesbarkeit empfiehlt es sich jedoch, nur Operationen auszuführen, die für die Schleife direkt relevant sind.
- Eine in der Initialisierungsanweisung deklarierte Variable ist nur innerhalb des Gültigkeitsbereichs der `for` Schleife sichtbar und wird bei Beendigung der Schleife freigegeben.
- Vergessen Sie nicht, dass die in der `initialization statement` deklarierte Variable während der Schleife sowie die in der `condition` geprüfte Variable geändert werden `condition` .

Beispiel für eine Schleife, die von 0 bis 10 zählt:

```
for (int counter = 0; counter <= 10; ++counter)
{
    std::cout << counter << '\n';
}
// counter is not accessible here (had value 11 at the end)
```

Erklärung der Codefragmente:

- `int counter = 0` initialisiert die Variable `counter` auf 0 (Diese Variable kann nur innerhalb der verwendet werden `for` die Schleife.)
- `counter <= 10` ist eine boolesche Bedingung, die prüft, ob der `counter` kleiner oder gleich 10 ist. Wenn er `true` , wird die Schleife ausgeführt. Wenn es `false` , endet die Schleife.
- `++counter` ist eine Inkrementierungsoperation, die den Wert des `counter` vor der nächsten Zustandsprüfung um 1 erhöht.

Wenn Sie alle Anweisungen leer lassen, können Sie eine Endlosschleife erstellen:

```
// infinite loop
for (;;)
    std::cout << "Never ending!\n";
```

Das Äquivalent der `while` Schleife des obigen ist:

```
// infinite loop
while (true)
    std::cout << "Never ending!\n";
```

Eine Endlosschleife kann jedoch immer noch verlassen werden, indem die Anweisungen `break` , `goto` oder `return` oder eine Ausnahme `goto` wird.

Das nächste gängige Beispiel für das Durchlaufen aller Elemente einer STL-Sammlung (z. B. eines `vector`) ohne Verwendung des Headers `<algorithm>` ist:

```
std::vector<std::string> names = {"Albert Einstein", "Stephen Hawking", "Michael Ellis"};
for(std::vector<std::string>::iterator it = names.begin(); it != names.end(); ++it) {
    std::cout << *it << std::endl;
}
```

While-Schleife

Eine `while` Schleife führt die Anweisungen wiederholt aus, bis die angegebene Bedingung als `false` ausgewertet wird. Diese Steueranweisung wird verwendet, wenn im Voraus nicht bekannt ist, wie oft ein Codeblock ausgeführt werden soll.

Um beispielsweise alle Zahlen von 0 bis 9 zu drucken, kann der folgende Code verwendet werden:

```
int i = 0;
while (i < 10)
{
    std::cout << i << " ";
    ++i; // Increment counter
}
std::cout << std::endl; // End of line; "0 1 2 3 4 5 6 7 8 9" is printed to the console
```

C ++ 17

Beachten Sie, dass die ersten zwei Anweisungen seit C ++ 17 kombiniert werden können

```
while (int i = 0; i < 10)
//... The rest is the same
```

Um eine Endlosschleife zu erstellen, kann das folgende Konstrukt verwendet werden:

```
while (true)
{
    // Do something forever (however, you can exit the loop by calling 'break'
}
```

Es gibt eine andere Variante von `while` Schleifen, nämlich das `do...while` Konstrukt. Weitere Informationen finden Sie im Do-while-Loop-Beispiel .

Deklaration von Variablen in Bedingungen

Im Zustand der `for` und `while` Schleifen darf auch ein Objekt deklariert werden. Dieses Objekt wird bis zum Ende der Schleife als Gültigkeitsbereich betrachtet und bleibt bei jeder Iteration der Schleife bestehen:

```
for (int i = 0; i < 5; ++i) {
    do_something(i);
}
// i is no longer in scope.

for (auto& a : some_container) {
```

```
    a.do_something();
}
// a is no longer in scope.

while(std::shared_ptr<Object> p = get_object()) {
   p->do_something();
}
// p is no longer in scope.
```

Es ist jedoch nicht erlaubt, dasselbe mit einer `do...while` Schleife auszuführen. Deklarieren Sie stattdessen die Variable vor der Schleife und schließen Sie (optional) sowohl die Variable als auch die Schleife in einen lokalen Gültigkeitsbereich ein, wenn die Variable nach dem Beenden der Schleife außerhalb des Gültigkeitsbereichs sein soll:

```
//This doesn't compile
do {
    s = do_something();
} while (short s > 0);

// Good
short s;
do {
    s = do_something();
} while (s > 0);
```

Dies liegt daran, dass der *Anweisungsteil* einer `do...while` Schleife (des Körpers der Schleife) ausgewertet wird, bevor der *Ausdrucksteil* (das `while`) erreicht ist. Daher werden Deklarationen im *Ausdruck* während der ersten Iteration des Befehls nicht sichtbar Schleife.

Do-while-Schleife

Eine *do-while*- Schleife ist einer *while*- Schleife sehr ähnlich, mit der Ausnahme, dass die Bedingung am Ende jedes Zyklus überprüft wird, nicht am Anfang. Die Schleife kann daher mindestens einmal ausgeführt werden.

Der folgende Code gibt `0` , da die Bedingung am Ende der ersten Iteration als `false` ausgewertet wird:

```
int i =0;
do
{
    std::cout << i;
    ++i; // Increment counter
}
while (i < 0);
std::cout << std::endl; // End of line; 0 is printed to the console
```

Hinweis: Vergessen Sie nicht das Semikolon am Ende von `while(condition);` , was im *do-while*-Konstrukt benötigt wird.

Im Gegensatz zur *Do-while*- Schleife wird Folgendes nicht gedruckt, da die Bedingung zu Beginn der ersten Iteration als `false` ausgewertet wird:

```
int i =0;
while (i < 0)
{
    std::cout << i;
    ++i; // Increment counter
}
std::cout << std::endl; // End of line; nothing is printed to the console
```

Hinweis: Eine *while*- Schleife kann beendet werden, ohne dass die Bedingung durch eine `break` , `goto` oder `return` `goto` falsch wird.

```
int i = 0;
do
{
    std::cout << i;
    ++i; // Increment counter
    if (i > 5)
    {
        break;
    }
}
while (true);
std::cout << std::endl; // End of line; 0 1 2 3 4 5 is printed to the console
```

Eine triviale *Do-while*- Schleife wird gelegentlich auch zum Schreiben von Makros verwendet, die einen eigenen Gültigkeitsbereich erfordern (in diesem Fall wird das nachstehende Semikolon aus der Makrodefinition weggelassen und muss vom Benutzer bereitgestellt werden)

```
#define BAD_MACRO(x) f1(x); f2(x); f3(x);

// Only the call to f1 is protected by the condition here
if (cond) BAD_MACRO(var);

#define GOOD_MACRO(x) do { f1(x); f2(x); f3(x); } while(0)

// All calls are protected here
if (cond) GOOD_MACRO(var);
```

Anweisungen zur Schleifensteuerung: Break and Continue

Schleifensteueranweisungen werden verwendet, um den Ablauf der Ausführung von seiner normalen Reihenfolge zu ändern. Wenn die Ausführung einen Bereich verlässt, werden alle automatischen Objekte, die in diesem Bereich erstellt wurden, zerstört. Das `break` und `continue` sind Schleifensteueranweisungen.

Die `break` Anweisung beendet eine Schleife ohne weitere Überlegung.

```
for (int i = 0; i < 10; i++)
{
    if (i == 4)
        break; // this will immediately exit our loop
    std::cout << i << '\n';
}
```

Der obige Code wird ausgedruckt:

```
1
2
3
```

Die `continue` Anweisung beendet die Schleife nicht sofort, sondern überspringt den Rest des Schleifenkörpers und geht zum Anfang der Schleife (einschließlich der Überprüfung der Bedingung).

```
for (int i = 0; i < 6; i++)
{
    if (i % 2 == 0) // evaluates to true if i is even
        continue; // this will immediately go back to the start of the loop
    /* the next line will only be reached if the above "continue" statement
       does not execute  */
    std::cout << i << " is an odd number\n";
}
```

Der obige Code wird ausgedruckt:

```
1 is an odd number
3 is an odd number
5 is an odd number
```

Da solche Kontrollflussänderungen für den Menschen manchmal schwer zu verstehen sind, werden `break` und `continue` sparsam eingesetzt. Eine einfachere Implementierung ist normalerweise einfacher zu lesen und zu verstehen. Beispielsweise könnte die erste `for` Schleife mit der obigen `break` folgendermaßen umgeschrieben werden:

```
for (int i = 0; i < 4; i++)
{
    std::cout << i << '\n';
}
```

Das zweite Beispiel mit `continue` kann wie folgt umgeschrieben werden:

```
for (int i = 0; i < 6; i++)
{
    if (i % 2 != 0) {
        std::cout << i << " is an odd number\n";
    }
}
```

Range-für über einen Unterbereich

Mithilfe von Bereichs-Basis-Schleifen können Sie einen Teilbereich eines bestimmten Containers oder einen anderen Bereich durchlaufen, indem Sie ein Proxy-Objekt generieren, das für Bereichs-basierte Schleifen geeignet ist.

```
template<class Iterator, class Sentinel=Iterator>
```

```
struct range_t {
  Iterator b;
  Sentinel e;
  Iterator begin() const { return b; }
  Sentinel end() const { return e; }
  bool empty() const { return begin()==end(); }
  range_t without_front( std::size_t count=1 ) const {
    if (std::is_same< std::random_access_iterator_tag, typename
std::iterator_traits<Iterator>::iterator_category >{} ) {
      count = (std::min)(std::size_t(std::distance(b,e)), count);
    }
    return {std::next(b, count), e};
  }
  range_t without_back( std::size_t count=1 ) const {
    if (std::is_same< std::random_access_iterator_tag, typename
std::iterator_traits<Iterator>::iterator_category >{} ) {
      count = (std::min)(std::size_t(std::distance(b,e)), count);
    }
    return {b, std::prev(e, count)};
  }
};

template<class Iterator, class Sentinel>
range_t<Iterator, Sentinel> range( Iterator b, Sentinal e ) {
  return {b,e};
}
template<class Iterable>
auto range( Iterable& r ) {
  using std::begin; using std::end;
  return range(begin(r),end(r));
}

template<class C>
auto except_first( C& c ) {
  auto r = range(c);
  if (r.empty()) return r;
  return r.without_front();
}
```

Jetzt können wir tun:

```
std::vector<int> v = {1,2,3,4};

for (auto i : except_first(v))
  std::cout << i << '\n';
```

und ausdrucken

```
2
3
4
```

`for(:range_expression)` Sie, dass die im `for(:range_expression)` der `for` Schleife generierten Zwischenobjekte zu dem Zeitpunkt abgelaufen sind, zu dem die `for` Schleife beginnt.

Schleifen online lesen: https://riptutorial.com/de/cplusplus/topic/589/schleifen

Kapitel 99: Schlüsselwort const

Syntax

- const Typ myVariable = initial; // deklariert eine const-Variable; kann nicht geändert werden
- const Type & myReference = meineVariable; // Deklariert einen Verweis auf eine const-Variable
- const Typ * myPointer = & meineVariable; // Deklariert einen Zeiger auf-const. Der Zeiger kann sich ändern, das zugrunde liegende Datenelement kann jedoch nicht über den Zeiger geändert werden
- Geben Sie * const myPointer = & myVariable; // Deklariert einen const-Zeiger. Der Zeiger kann nicht neu zugewiesen werden, um auf etwas anderes zu verweisen, das darunterliegende Datenelement kann jedoch geändert werden
- const Typ * const myPointer = & meineVariable; // Deklariert einen konstanten Zeiger auf eine konstante.

Bemerkungen

Eine Variable als markiert `const` kann nicht [1] geändert werden. Wenn Sie versuchen, keine Nicht-Konstanten-Operationen aufzurufen, wird ein Compiler-Fehler ausgegeben.

1: Nun, es kann durch `const_cast` geändert werden, aber das sollte man fast nie verwenden

Examples

Lokale Variablen definieren

Erklärung und Verwendung.

```
// a is const int, so it can't be changed
const int a = 15;
a = 12;           // Error: can't assign new value to const variable
a += 1;           // Error: can't assign new value to const variable
```

Bindung von Referenzen und Verweisen

```
int &b = a;       // Error: can't bind non-const reference to const variable
const int &c = a; // OK; c is a const reference

int *d = &a;      // Error: can't bind pointer-to-non-const to const variable
const int *e = &a // OK; e is a pointer-to-const

int f = 0;
e = &f;           // OK; e is a non-const pointer-to-const,
                  // which means that it can be rebound to new int* or const int*

*e = 1            // Error: e is a pointer-to-const which means that
                  // the value it points to can't be changed through dereferencing e
```

```
int *g = &f;
*g = 1;           // OK; this value still can be changed through dereferencing
                  // a pointer-not-to-const
```

Konstanten

```
int a = 0, b = 2;

const int* pA = &a; // pointer-to-const. `a` can't be changed through this
int* const pB = &a; // const pointer. `a` can be changed, but this pointer can't.
const int* const pC = &a; // const pointer-to-const.

//Error: Cannot assign to a const reference
*pA = b;

pA = &b;

*pB = b;

//Error: Cannot assign to const pointer
pB = &b;

//Error: Cannot assign to a const reference
*pC = b;

//Error: Cannot assign to const pointer
pC = &b;
```

Const-Member-Funktionen

Member-Funktionen einer Klasse können als `const` deklariert werden. Dies sagt dem Compiler und zukünftigen Lesern, dass diese Funktion das Objekt nicht ändert:

```
class MyClass
{
private:
    int myInt_;
public:
    int myInt() const { return myInt_; }
    void setMyInt(int myInt) { myInt_ = myInt; }
};
```

In einer `const` Member-Funktion ist `this` Zeiger praktisch eine `const MyClass *` anstelle einer `MyClass *` . Das bedeutet, dass Sie keine Mitgliedsvariablen innerhalb der Funktion ändern können. Der Compiler gibt eine Warnung aus. `setMyInt` konnte daher nicht als `const` deklariert werden.

Sie sollten `const` fast immer als `const` markieren, wenn dies möglich ist. Nur `const` Member - Funktionen können auf einem heißen `const MyClass` .

`static` Methoden können nicht als `const` deklariert werden. Dies liegt daran, dass eine statische Methode zu einer Klasse gehört und für ein Objekt nicht aufgerufen wird. Daher können die internen Variablen des Objekts niemals geändert werden. Es wäre also überflüssig, `static` Methoden als `const` zu deklarieren.

Vermeidung der Duplizierung von Code in const- und non-const-Getter-Methoden.

In C ++ - Methoden, die sich nur durch `const` Qualifikationsmerkmal unterscheiden, kann es zu einer Überlastung kommen. Manchmal sind zwei Getter-Versionen erforderlich, die einen Verweis auf ein Mitglied zurückgeben.

Sei `Foo` eine Klasse mit zwei Methoden, die identische Operationen ausführen und einen Verweis auf ein Objekt vom Typ `Bar` :

```
class Foo
{
public:
    Bar& GetBar(/* some arguments */)
    {
        /* some calculations */
        return bar;
    }

    const Bar& GetBar(/* some arguments */) const
    {
        /* some calculations */
        return bar;
    }

    // ...
};
```

Der einzige Unterschied zwischen ihnen besteht darin, dass eine Methode eine Nicht-Konstante ist und eine Nicht-Konstantenreferenz zurückgibt (die zum Ändern des Objekts verwendet werden kann) und die zweite Konstante und eine Konstantenreferenz zurückgibt.

Um die Code-Duplizierung zu vermeiden, besteht die Versuchung, eine Methode von einer anderen aufzurufen. Die non-const-Methode kann jedoch nicht von der const-Methode aufgerufen werden. Die const-Methode können wir jedoch von einer nicht-const-Methode aus aufrufen. Dies erfordert die Verwendung von 'const_cast', um das const-Qualifikationsmerkmal zu entfernen.

Die Lösung ist:

```
struct Foo
{
    Bar& GetBar(/*arguments*/)
    {
        return const_cast<Bar&>(const_cast<const Foo*>(this)->GetBar(/*arguments*/));
    }

    const Bar& GetBar(/*arguments*/) const
    {
        /* some calculations */
        return foo;
    }
};
```

Im obigen Code rufen wir die const-Version von `GetBar` von der nicht-const- `GetBar` indem wir dies

in const-Typ `const_cast<const Foo*>(this)` : `const_cast<const Foo*>(this)` . Da wir die const-Methode von non-const aufrufen, ist das Objekt selbst nicht-const und das Verwerfen der const ist erlaubt.

Untersuchen Sie das folgende, vollständigere Beispiel:

```
#include <iostream>

class Student
{
public:
    char& GetScore(bool midterm)
    {
        return const_cast<char&>(const_cast<const Student*>(this)->GetScore(midterm));
    }

    const char& GetScore(bool midterm) const
    {
        if (midterm)
        {
            return midtermScore;
        }
        else
        {
            return finalScore;
        }
    }

private:
    char midtermScore;
    char finalScore;
};

int main()
{
    // non-const object
    Student a;
    // We can assign to the reference. Non-const version of GetScore is called
    a.GetScore(true) = 'B';
    a.GetScore(false) = 'A';

    // const object
    const Student b(a);
    // We still can call GetScore method of const object,
    // because we have overloaded const version of GetScore
    std::cout << b.GetScore(true) << b.GetScore(false) << '\n';
}
```

Schlüsselwort const online lesen: https://riptutorial.com/de/cplusplus/topic/2386/schlusselwort-const

Kapitel 100: Schlüsselwörter

Einführung

Schlüsselwörter haben eine durch den C ++ - Standard definierte feste Bedeutung und können nicht als Bezeichner verwendet werden. Es ist nicht zulässig, Schlüsselwörter mithilfe des Präprozessors in einer Übersetzungseinheit neu zu definieren, die einen Standard-Bibliothekskopf enthält. Schlüsselwörter verlieren jedoch ihre besondere Bedeutung in Attributen.

Syntax

- asm (*String-Literal*);
- noexcept (*Ausdruck*) // Bedeutung 1
- noexcept (*konstanter Ausdruck*) // Bedeutung 2
- noexcept // bedeutung 2
- sizeof *unary-expression*
- sizeof (*Typ-ID*)
- sizeof ... (*Bezeichner*) // seit C ++ 11
- Type - Name *verschachtelt-name-Bezeichner Kennung* // 1 Bedeutung
- Typname Vorlage für *geschachtelte Namen* (optional) (*opt*) *simple-template-id* // Bedeutung 1
- Typname *Kennung (opt)* // Bedeutung 2
- Typname ... *Bezeichner* (*Opt*) // Bedeutung 2; seit C ++ 11
- *Bezeichner* Typname (*opt*) = *Typ-ID* // Bedeutung 2
- template < *template-parameter-list* > typename ... (*opt*) *Bezeichner* (*opt*) // Bedeutung 3
- template <*template-Parameter-Liste*> Typname *Kennung (opt)* = *id-Ausdruck* // Bedeutung 3

Bemerkungen

Die vollständige Liste der Keywords lautet wie folgt:

- `alignas` (seit C ++ 11)
- `alignof` (seit C ++ 11)
- `asm`
- `auto` : seit C ++ 11 vor C ++ 11
- `bool`
- `break`
- `case`
- `catch`
- `char`
- `char16_t` (seit C ++ 11)
- `char32_t` (seit C ++ 11)
- `class`
- `const`
- `constexpr` (seit C ++ 11)
- `const_cast`

- `continue`
- `decltype` (seit C ++ 11)
- `default`
- `delete` für die Speicherverwaltung für Funktionen (seit C ++ 11)
- `do`
- `double`
- `dynamic_cast`
- `else`
- `enum`
- `explicit`
- `export`
- `extern` als Deklarationsspezifizierer , in der Verknüpfungsspezifikation für Vorlagen
- `false`
- `float`
- `for`
- `friend`
- `goto`
- `if`
- `inline` für Funktionen , für Namespaces (seit C ++ 11), für Variablen (seit C ++ 17)
- `int`
- `long`
- `mutable`
- `namespace`
- `new`
- `noexcept` (seit C ++ 11)
- `nullptr` (seit C ++ 11)
- `operator`
- `private`
- `protected`
- `public`
- `register`
- `reinterpret_cast`
- `return`
- `short`
- `signed`
- `sizeof`
- `static`
- `static_assert` (seit C ++ 11)
- `static_cast`
- `struct`
- `switch`
- `template`
- `this`
- `thread_local` (seit C ++ 11)
- `throw`
- `true`
- `try`
- `typedef`
- `typeid`
- `typename`
- `union`
- `unsigned`
- `using` , um einen Namen erneut zu deklarieren, einen Aliasnamen oder einen Aliasnamen zu definieren

- `virtual` für Funktionen , für Basisklassen
- `void`
- `volatile`
- `wchar_t`
- `while`

Das Token `final` und `override` sind keine Schlüsselwörter. Sie können als Bezeichner verwendet werden und haben nur in bestimmten Zusammenhängen eine besondere Bedeutung.

Die Token `and` , `and_eq` , `bitand` , `bitor` , `compl` , `not` , `not_eq` `or` `or_eq` , `xor` und `xor_eq` sind alternative Schreibweisen von `&&` , `&=` , `&` , `|` , `~` `!` , `!=` , `||` , `|=` , `^` und `^=` . Der Standard behandelt sie nicht als Schlüsselwörter, aber sie sind Schlüsselwörter für alle Absichten und Zwecke, da es unmöglich ist, sie neu zu definieren oder sie zu verwenden, um etwas anderes als die von ihnen repräsentierten Operatoren zu bedeuten.

Die folgenden Themen enthalten ausführliche Erklärungen zu vielen Schlüsselwörtern in C ++, die grundlegenden Zwecken dienen, z. B. zum Benennen von Basistypen oder zum Steuern des Ausführungsflusses.

- Grundtyp-Schlüsselwörter
- Ablaufsteuerung
- Iteration
- Wörtliche Schlüsselwörter
- Geben Sie Schlüsselwörter ein
- Variablendeklarationsschlüsselwörter
- Klassen / Strukturen
- Speicherklassenspezifizierer

Examples

asm

Das Schlüsselwort `asm` akzeptiert einen einzelnen Operanden, der ein String-Literal sein muss. Es hat eine implementierungsdefinierte Bedeutung, wird jedoch normalerweise an den Assembler der Implementierung übergeben, wobei die Ausgabe des Assemblers in die Übersetzungseinheit integriert wird.

Die `asm` Anweisung ist eine *Definition* und kein *Ausdruck* . Daher kann sie entweder im Blockbereich oder im Namespace-Bereich (einschließlich des globalen Bereichs) stehen. Da die Inline-Assembly jedoch nicht durch die Regeln der C ++ - Sprache eingeschränkt werden kann, erscheint `asm` möglicherweise nicht in einer `constexpr` Funktion.

Beispiel:

```
[[noreturn]] void halt_system() {
    asm("hlt");
}
```

explizit

1. Bei Anwendung auf einen Konstruktor mit einem einzigen Argument wird verhindert, dass dieser Konstruktor für implizite Konvertierungen verwendet wird.

```
class MyVector {
  public:
    explicit MyVector(uint64_t size);
};
MyVector v1(100);  // ok
uint64_t len1 = 100;
MyVector v2{len1}; // ok, len1 is uint64_t
int len2 = 100;
MyVector v3{len2}; // ill-formed, implicit conversion from int to uint64_t
```

   Seit C ++ 11 Initializer-Listen eingeführt hat, kann `explicit` in C ++ 11 und höher auf einen Konstruktor mit einer beliebigen Anzahl von Argumenten angewendet werden, die dieselbe Bedeutung haben wie im Fall eines einzelnen Arguments.

```
struct S {
    explicit S(int x, int y);
};
S f() {
    return {12, 34};  // ill-formed
    return S{12, 34}; // ok
}
```

C ++ 11

2. Bei Anwendung auf eine Konvertierungsfunktion wird verhindert, dass diese Konvertierungsfunktion für implizite Konvertierungen verwendet wird.

```
class C {
    const int x;
  public:
    C(int x) : x(x) {}
    explicit operator int() { return x; }
};
C c(42);
int x = c;                    // ill-formed
int y = static_cast<int>(c); // ok; explicit conversion
```

noexcept

C ++ 11

1. Ein unärer Operator, der bestimmt, ob die Auswertung seines Operanden eine Ausnahme auslösen kann. Beachten Sie, dass die Körper der aufgerufenen Funktionen nicht untersucht

werden, sodass `noexcept` falsche Negative `noexcept` kann. Der Operand wird nicht ausgewertet.

```
#include <iostream>
#include <stdexcept>
void foo() { throw std::runtime_error("oops"); }
void bar() {}
struct S {};
int main() {
    std::cout << noexcept(foo()) << '\n'; // prints 0
    std::cout << noexcept(bar()) << '\n'; // prints 0
    std::cout << noexcept(1 + 1) << '\n'; // prints 1
    std::cout << noexcept(S()) << '\n';   // prints 1
}
```

Auch wenn `bar()` niemals eine Ausnahme `noexcept(bar())` kann, ist `noexcept(bar())` immer noch falsch, da die Tatsache, dass `bar()` keine Ausnahme `noexcept(bar())` kann, nicht explizit angegeben wurde.

2. Gibt bei der Deklaration einer Funktion an, ob die Funktion eine Ausnahme weitergeben kann. Allein erklärt es, dass die Funktion keine Ausnahme weitergeben kann. Mit einem Argument in Klammern erklärt es, dass die Funktion abhängig vom Wahrheitswert des Arguments eine Ausnahme ausgeben kann oder nicht.

```
void f1() { throw std::runtime_error("oops"); }
void f2() noexcept(false) { throw std::runtime_error("oops"); }
void f3() {}
void f4() noexcept {}
void f5() noexcept(true) {}
void f6() noexcept {
    try {
        f1();
    } catch (const std::runtime_error&) {}
}
```

In diesem Beispiel haben wir erklärt, dass `f4` , `f5` und `f6` keine Ausnahmen verbreiten können. (Obwohl eine Ausnahme während der Ausführung von `f6` ausgelöst werden kann, wird sie abgefangen und darf sich nicht aus der Funktion ausbreiten.) Wir haben erklärt, dass `f2` eine Ausnahme `f2` kann. Wenn der `noexcept` weggelassen wird, entspricht er `noexcept(false)` haben wir implizit erklärt, dass `f1` und `f3` Ausnahmen ausbreiten können, obwohl Ausnahmen während der Ausführung von `f3` nicht wirklich ausgelöst werden können.

C ++ 17

Ob eine Funktion keine `noexcept` ist oder nicht, ist Teil des Funktionstyps: Im obigen Beispiel unterscheiden sich `f1` , `f2` und `f3` von `f4` , `f5` und `f6` . Daher ist `noexcept` auch für Funktionszeiger, Vorlagenargumente usw. von Bedeutung.

```
void g1() {}
void g2() noexcept {}
void (*p1)() noexcept = &g1; // ill-formed, since g1 is not noexcept
void (*p2)() noexcept = &g2; // ok; types match
void (*p3)() = &g1;          // ok; types match
```

```
void (*p4)() = &g2;          // ok; implicit conversion
```

Modellname

1. Wenn er von einem qualifizierten Namen gefolgt wird, gibt `typename` an, dass es sich um den Namen eines Typs handelt. Dies ist häufig in Vorlagen erforderlich, insbesondere wenn der geschachtelte Namensbezeichner ein von der aktuellen Instanziierung abweichender Typ ist. In diesem Beispiel `std::decay<T>` ist abhängig von dem Template - Parametern `T` , so um den verschachtelte Art zu nennen `type` , wir die gesamten qualifizierten Namen mit Präfix müssen `typename` . Weitere Informationen finden Sie unter Wo und warum muss ich die Schlüsselwörter "template" und "typename" eingeben?

   ```
   template <class T>
   auto decay_copy(T&& r) -> typename std::decay<T>::type;
   ```

2. Führt einen Typparameter in die Deklaration einer Vorlage ein . In diesem Zusammenhang ist es mit der `class` austauschbar.

   ```
   template <typename T>
   const T& min(const T& x, const T& y) {
       return b < a ? b : a;
   }
   ```

C ++ 17

3. `typename` kann auch verwendet werden, wenn ein template-Template-Parameter deklariert wird , der wie der `class` dem Namen des Parameters vorangeht.

   ```
   template <template <class T> typename U>
   void f() {
       U<int>::do_it();
       U<double>::do_it();
   }
   ```

Größe von

Ein unärer Operator, der die Größe seines Operanden in Bytes ermittelt, der entweder ein Ausdruck oder ein Typ sein kann. Wenn der Operand ein Ausdruck ist, wird er nicht ausgewertet. Die Größe ist ein konstanter Ausdruck des Typs `std::size_t` .

Wenn der Operand ein Typ ist, muss er in Klammern gesetzt werden.

- Es ist nicht `sizeof` , `sizeof` auf einen Funktionstyp anzuwenden.
- Es ist nicht `sizeof` , `sizeof` auf einen unvollständigen Typ, einschließlich `void` , anzuwenden.
- Wenn sizeof auf einen Referenztyp `T&` oder `T&&` angewendet wird, entspricht es `sizeof(T)` .
- Wenn `sizeof` auf einen Klassentyp angewendet wird, gibt es die Anzahl der Bytes in einem vollständigen Objekt dieses Typs an, einschließlich der Auffüllbytes in der Mitte oder am Ende. Daher kann ein `sizeof` Ausdruck niemals den Wert 0 haben. Weitere Informationen finden Sie im Layout der Objekttypen .

- Die Typen `char` , `signed char` und `unsigned char` haben eine Größe von 1. Umgekehrt wird ein Byte als die Menge Speicher definiert, die zum Speichern eines `char` Objekts erforderlich ist. Es muss nicht unbedingt 8 Bit bedeuten, da einige Systeme haben `char` Objekte länger als 8 Bit.

Wenn *expr* ein Ausdruck ist, ist `sizeof(` *expr* `)` äquivalent zu `sizeof(T)` wobei `T` der Typ von *expr ist.*

```
int a[100];
std::cout << "The number of bytes in `a` is: " << sizeof a;
memset(a, 0, sizeof a); // zeroes out the array
```

C ++ 11

Der Operator `sizeof...` gibt die Anzahl der Elemente in einem Parameterpaket an.

```
template <class... T>
void f(T&&...) {
    std::cout << "f was called with " << sizeof...(T) << " arguments\n";
}
```

Unterschiedliche Schlüsselwörter

C ++ ungültig machen

1. Bei Verwendung als Funktionsrückgabetyp gibt das Schlüsselwort void an, dass die Funktion keinen Wert zurückgibt. Wenn void für die Parameterliste einer Funktion verwendet wird, gibt void an, dass die Funktion keine Parameter annimmt. Bei der Deklaration eines Zeigers gibt void an, dass der Zeiger "universal" ist.

2. Wenn der Zeigertyp void * ist, kann der Zeiger auf eine beliebige Variable zeigen, die nicht mit dem Schlüsselwort const oder volatile deklariert ist. Ein ungültiger Zeiger kann nicht dereferenziert werden, wenn er nicht in einen anderen Typ umgewandelt wird. Ein ungültiger Zeiger kann in einen beliebigen anderen Datenzeiger-Typ umgewandelt werden.

3. Ein ungültiger Zeiger kann auf eine Funktion zeigen, aber nicht auf ein Klassenmitglied in C ++.

```
void vobject;   // C2182
void *pv;   // okay
int *pint; int i;
int main() {
pv = &i;
   // Cast optional in C required in C++
pint = (int *)pv;
```

Flüchtiges C ++

1. Ein Typqualifizierer, mit dem Sie angeben können, dass ein Objekt im Programm von der Hardware geändert werden kann.

```
volatile declarator ;
```

virtuelles C ++

1. Das virtuelle Schlüsselwort deklariert eine virtuelle Funktion oder eine virtuelle Basisklasse.

```
virtual [type-specifiers] member-function-declarator
virtual [access-specifier] base-class-name
```

Parameter

1. **Typspezifizierer** Gibt den Rückgabetyp der virtuellen Memberfunktion an.
2. **member-function-declarator Deklariert** eine Member-Funktion.
3. **Zugriffsspezifizierer** Definiert die Zugriffsebene für die Basisklasse (öffentlich, geschützt oder privat). Kann vor oder nach dem virtuellen Schlüsselwort angezeigt werden.
4. **Basisklassenname** Gibt einen zuvor deklarierten Klassentyp an

dieser Zeiger

1. Dieser Zeiger ist ein Zeiger, auf den nur innerhalb der nicht statischen Memberfunktionen einer Klasse, Struktur oder eines Unionstyps zugegriffen werden kann. Es zeigt auf das Objekt, für das die Memberfunktion aufgerufen wird. Statische Memberfunktionen haben diesen Zeiger nicht.

```
this->member-identifier
```

Der Zeiger eines Objekts ist nicht Teil des Objekts selbst; Sie wird nicht im Ergebnis einer Anweisung sizeof für das Objekt berücksichtigt. Wenn stattdessen eine nicht statische Member-Funktion für ein Objekt aufgerufen wird, wird die Adresse des Objekts vom Compiler als verstecktes Argument an die Funktion übergeben. Zum Beispiel den folgenden Funktionsaufruf:

```
myDate.setMonth( 3 );

can be interpreted this way:


setMonth( &myDate, 3 );

The object's address is available from within the member function as the this pointer. Most
uses of this are implicit. It is legal, though unnecessary, to explicitly use this when
referring to members of the class. For example:


void Date::setMonth( int mn )
{
   month = mn;             // These three statements
   this->month = mn;       // are equivalent
   (*this).month = mn;
}
```

```
The expression *this is commonly used to return the current object from a member function:


return *this;

The this pointer is also used to guard against self-reference:


if (&Object != this) {
// do not execute in cases of self-reference
```

Anweisungen versuchen, werfen und abfangen (C ++)

1. Um die Ausnahmebehandlung in C ++ zu implementieren, verwenden Sie try, throw und catch-Ausdrücke.
2. Verwenden Sie zunächst einen try-Block, um eine oder mehrere Anweisungen einzuschließen, die eine Ausnahme auslösen können.
3. Ein Wurfausdruck signalisiert, dass in einem try-Block eine außergewöhnliche Bedingung - oft ein Fehler - aufgetreten ist. Sie können ein Objekt eines beliebigen Typs als Operanden eines Wurfausdrucks verwenden. Normalerweise wird dieses Objekt verwendet, um Informationen zu dem Fehler zu übermitteln. In den meisten Fällen wird empfohlen, die std :: ausnahmeklasse oder eine der abgeleiteten Klassen zu verwenden, die in der Standardbibliothek definiert sind. Wenn eine davon nicht geeignet ist, empfehlen wir Ihnen, Ihre eigene Exception-Klasse von std :: exception abzuleiten.
4. Implementieren Sie einen oder mehrere catch-Blöcke unmittelbar nach einem try-Block, um möglicherweise ausgelöste Ausnahmen zu behandeln. Jeder catch-Block gibt den Typ der Ausnahme an, die er behandeln kann.

```
    MyData md;
try {
   // Code that could throw an exception
   md = GetNetworkResource();
}
catch (const networkIOException& e) {
   // Code that executes when an exception of type
   // networkIOException is thrown in the try block
   // ...
   // Log error message in the exception object
   cerr << e.what();
}
catch (const myDataFormatException& e) {
   // Code that handles another exception type
   // ...
   cerr << e.what();
}

// The following syntax shows a throw expression
MyData GetNetworkResource()
{
   // ...
   if (IOSuccess == false)
      throw networkIOException("Unable to connect");
   // ...
   if (readError)
```

```
      throw myDataFormatException("Format error");
   // ...
}
```

Der Code nach der try-Klausel ist der überwachte Codeabschnitt. Der Wurfausdruck wirft - das heißt, löst eine Ausnahme aus. Der Codeblock nach der catch-Klausel ist der Ausnahmehandler. Dies ist der Handler, der die Ausnahme abfängt, die ausgelöst wird, wenn die Typen in den Wurf- und Fangausdrücken kompatibel sind.

```
    try {
   throw CSomeOtherException();
}
catch(...) {
   // Catch all exceptions – dangerous!!!
   // Respond (perhaps only partially) to the exception, then
   // re-throw to pass the exception to some other handler
   // ...
   throw;
}
```

Freund (C ++)

1. Unter bestimmten Umständen ist es günstiger, auf Mitgliedsebene Zugriff auf Funktionen zu gewähren, die keine Mitglieder einer Klasse sind, oder auf alle Mitglieder einer separaten Klasse. Nur der Klassenimplementierer kann angeben, wer seine Freunde sind. Eine Funktion oder Klasse kann sich nicht als Freund einer Klasse deklarieren. Verwenden Sie in einer Klassendefinition das Schlüsselwort friend und den Namen einer Nichtmitgliedsfunktion oder einer anderen Klasse, um diesem Zugriff auf die privaten und geschützten Mitglieder Ihrer Klasse zu gewähren. In einer Vorlagendefinition kann ein Typparameter als Freund deklariert werden.

2. Wenn Sie eine Friend-Funktion deklarieren, die zuvor nicht deklariert wurde, wird diese Funktion in den umschließenden nicht-Klassenbereich exportiert.

```
class friend F
friend F;
class ForwardDeclared;// Class name is known.
class HasFriends
{
   friend int ForwardDeclared::IsAFriend();// C2039 error expected
};
```

Freundesfunktionen

1. Eine Friend-Funktion ist eine Funktion, die kein Mitglied einer Klasse ist, aber Zugriff auf die privaten und geschützten Member der Klasse hat. Freunde-Funktionen werden nicht als Klassenmitglieder betrachtet. Es handelt sich hierbei um normale externe Funktionen, die über spezielle Zugriffsrechte verfügen.

2. Freunde sind nicht im Gültigkeitsbereich der Klasse und werden nicht mit den Elementauswahloperatoren (. Und ->) aufgerufen, es sei denn, sie sind Mitglieder einer

anderen Klasse.

3. Eine Friend-Funktion wird von der Klasse deklariert, die den Zugriff gewährt. Die Friend-Deklaration kann an beliebiger Stelle in der Klassendeklaration platziert werden. Es ist von den Schlüsselwörtern für die Zugriffskontrolle nicht betroffen.

```
#include <iostream>

using namespace std;
class Point
{
    friend void ChangePrivate( Point & );
public:
    Point( void ) : m_i(0) {}
    void PrintPrivate( void ){cout << m_i << endl; }

private:
int m_i;
};

void ChangePrivate ( Point &i ) { i.m_i++; }

int main()
{
   Point sPoint;
   sPoint.PrintPrivate();
   ChangePrivate(sPoint);
   sPoint.PrintPrivate();
    // Output: 0
           1
}
```

Klassenmitglieder als Freunde

```
class B;

class A {
public:
   int Func1( B& b );

private:
   int Func2( B& b );
};

class B {
private:
int _b;

   // A::Func1 is a friend function to class B
   // so A::Func1 has access to all members of B
   friend int A::Func1( B& );
};

int A::Func1( B& b ) { return b._b; }   // OK
int A::Func2( B& b ) { return b._b; }   // C2248
```

Schlüsselwörter online lesen: https://riptutorial.com/de/cplusplus/topic/4891/schlusselworter

Kapitel 101: Semantik verschieben

Examples

Semantik verschieben

Verschiebesemantik ist eine Möglichkeit, ein Objekt in C ++ zu einem anderen zu verschieben. Dazu leeren wir das alte Objekt und legen alles in das neue Objekt.

Dazu müssen wir verstehen, was ein Referenzwert ist. Eine rvalue-Referenz (`T&&` wobei T der Objekttyp ist) unterscheidet sich nicht wesentlich von einer normalen Referenz (`T&` , jetzt lvalue-referenzen genannt). Sie fungieren jedoch als zwei verschiedene Typen, und so können Konstruktoren oder Funktionen erstellt werden, die den einen oder den anderen Typ annehmen, was bei der Umzugssemantik erforderlich ist.

Der Grund, warum wir zwei unterschiedliche Typen benötigen, besteht darin, zwei unterschiedliche Verhaltensweisen anzugeben. Lvalue-Referenzkonstruktoren beziehen sich auf das Kopieren, während sich rvalue-Referenzkonstruktoren auf das Verschieben beziehen.

Um ein Objekt zu verschieben, verwenden wir `std::move(obj)` . Diese Funktion gibt eine rvalue-Referenz auf das Objekt zurück, sodass wir die Daten von diesem Objekt in ein neues stehlen können. Dazu gibt es mehrere Möglichkeiten, die im Folgenden beschrieben werden.

Es ist wichtig zu beachten, dass durch die Verwendung von `std::move` nur eine rvalue-Referenz erstellt wird. Mit anderen Worten ändert die Anweisung `std::move(obj)` den Inhalt von obj nicht, während `auto obj2 = std::move(obj)` (möglicherweise) dies tut.

Konstruktor verschieben

Angenommen, wir haben diesen Code-Ausschnitt.

```
class A {
public:
    int a;
    int b;

    A(const A &other) {
        this->a = other.a;
        this->b = other.b;
    }
};
```

Um einen Kopierkonstruktor zu erstellen, dh um eine Funktion zu erstellen, die ein Objekt kopiert und ein neues erstellt, wählen wir normalerweise die oben gezeigte Syntax. Wir haben einen Konstruktor für A, der einen Verweis auf ein anderes Objekt vom Typ A verwendet und wir würden das Objekt manuell in die Methode kopieren.

Alternativ hätten wir `A(const A &) = default;` schreiben können `A(const A &) = default;` die

automatisch über alle Elemente kopiert wird, wobei der Copy-Konstruktor verwendet wird.

Um einen Bewegungskonstruktor zu erstellen, verwenden wir jedoch anstelle des Werts eine Referenz, wie hier.

```
class Wallet {
public:
    int nrOfDollars;

    Wallet() = default; //default ctor

    Wallet(Wallet &&other) {
        this->nrOfDollars = other.nrOfDollars;
        other.nrOfDollars = 0;
    }
};
```

Bitte beachten Sie, dass wir die alten Werte auf `zero` . Der Standardkonstruktor für Verschiebungen (`Wallet(Wallet&&) = default;`) kopiert den Wert von `nrOfDollars` , da es sich um einen POD handelt.

Da die Umzugs-Semantik so gestaltet ist, dass ein Zustand "Diebstahl" von der ursprünglichen Instanz ermöglicht wird, muss berücksichtigt werden, wie die ursprüngliche Instanz nach diesem Diebstahl aussehen sollte. In diesem Fall hätten wir, wenn wir den Wert nicht auf null ändern würden, den Betrag des Dollars verdoppelt.

```
Wallet a;
a.nrOfDollars = 1;
Wallet b (std::move(a)); //calling B(B&& other);
std::cout << a.nrOfDollars << std::endl; //0
std::cout << b.nrOfDollars << std::endl; //1
```

So haben wir aus einem alten Objekt ein Objekt konstruiert.

Während das obige Beispiel ein einfaches Beispiel ist, zeigt es, was der Bewegungskonstruktor tun soll. In komplexeren Fällen, beispielsweise beim Ressourcenmanagement, ist dies sinnvoller.

```
    // Manages operations involving a specified type.
    // Owns a helper on the heap, and one in its memory (presumably on the stack).
    // Both helpers are DefaultConstructible, CopyConstructible, and MoveConstructible.
    template<typename T,
             template<typename> typename HeapHelper,
             template<typename> typename StackHelper>
    class OperationsManager {
        using MyType = OperationsManager<T, HeapHelper, StackHelper>;

        HeapHelper<T>* h_helper;
        StackHelper<T> s_helper;
        // ...

      public:
        // Default constructor & Rule of Five.
        OperationsManager() : h_helper(new HeapHelper<T>) {}
        OperationsManager(const MyType& other)
```

```
          : h_helper(new HeapHelper<T>(*other.h_helper)), s_helper(other.s_helper) {}
        MyType& operator=(MyType copy) {
            swap(*this, copy);
            return *this;
        }
        ~OperationsManager() {
            if (h_helper) { delete h_helper; }
        }

        // Move constructor (without swap()).
        // Takes other's HeapHelper<T>*.
        // Takes other's StackHelper<T>, by forcing the use of StackHelper<T>'s move
constructor.
        // Replaces other's HeapHelper<T>* with nullptr, to keep other from deleting our shiny
        //  new helper when it's destroyed.
        OperationsManager(MyType&& other) noexcept
          : h_helper(other.h_helper),
            s_helper(std::move(other.s_helper)) {
            other.h_helper = nullptr;
        }

        // Move constructor (with swap()).
        // Places our members in the condition we want other's to be in, then switches members
        //  with other.
        // OperationsManager(MyType&& other) noexcept : h_helper(nullptr) {
        //     swap(*this, other);
        // }

        // Copy/move helper.
        friend void swap(MyType& left, MyType& right) noexcept {
            std::swap(left.h_helper, right.h_helper);
            std::swap(left.s_helper, right.s_helper);
        }
    };
```

Zuordnung verschieben

Ähnlich wie wir einem Objekt einen Wert mit einer lvalue-Referenz zuweisen und kopieren können, können wir die Werte auch von einem Objekt zu einem anderen verschieben, ohne einen neuen zu erstellen. Wir nennen diese Umzugszuordnung. Wir verschieben die Werte von einem Objekt zu einem anderen vorhandenen Objekt.

Dazu müssen wir `operator =` überladen, nicht so, dass eine lvalue-Referenz wie bei der Zuweisung von Kopien erforderlich ist, sondern eine rvalue-Referenz.

```
class A {
    int a;
    A& operator= (A&& other) {
        this->a = other.a;
        other.a = 0;
        return *this;
    }
};
```

Dies ist die typische Syntax zum Definieren der Bewegungszuweisung. Wir überladen `operator =` damit wir ihm eine rvalue-Referenz geben können, die er einem anderen Objekt zuweisen kann.

```
A a;
a.a = 1;
A b;
b = std::move(a); //calling A& operator= (A&& other)
std::cout << a.a << std::endl; //0
std::cout << b.a << std::endl; //1
```

Daher können wir verschieben, ein Objekt einem anderen Objekt zuweisen.

Verwenden von std :: move, um die Komplexität von O (n²) nach O (n) zu reduzieren

Mit C ++ 11 wurde die Unterstützung für Kernsprachen und Standardbibliotheken zum **Verschieben** eines Objekts eingeführt. Die Idee ist , dass , wenn ein Objekt *o* eine temporäres ist und man eine logische Kopie will, dann sicher seiner nur pilfer *o* ,s Ressourcen, wie zum Beispiel eines dynamisch zugewiesenen Puffer, so dass *o* logisch leer , aber immer noch zerstörbar und kopierbar.

Die Hauptsprachenunterstützung ist hauptsächlich

- Der **Builder für** den **rvalue-Referenztyp** `&&` , z. B. `std::string&&` ist ein rvalue-Verweis auf einen `std::string` , der darauf hinweist, dass das referenzierte Objekt ein temporäres Objekt ist, dessen Ressourcen nur gestohlen (dh verschoben) werden können.
- spezielle Unterstützung für einen **Bewegungskonstruktor** `T( T&& )` , der Ressourcen effizient aus dem angegebenen anderen Objekt verschieben soll, anstatt diese Ressourcen tatsächlich zu kopieren, und
- spezielle Unterstützung für einen **Verschiebungszuweisungsoperator** `auto operator=(T&&) -> T&` , der auch von der Quelle `auto operator=(T&&) -> T&` soll.

Die Standard-Bibliotheksunterstützung ist hauptsächlich die Funktionsvorlage `std::move` aus dem Header `<utility>` . Diese Funktion erzeugt eine rvalue-Referenz auf das angegebene Objekt, um anzuzeigen, dass es aus diesem Objekt verschoben werden kann, als wäre es ein temporärer.

Für einen Container ist das tatsächliche Kopieren typischerweise eine O (*n*) -Komplexität, wobei *n* die Anzahl der Elemente im Container ist, während das Verschieben eine konstante Zeit von O (1) ist. Und für einen Algorithmus, der logisch Kopien , die Container *n* - mal, dies die Komplexität des in der Regel unpraktisch O reduzieren können *(n* ²) nur lineare O *(n).*

In seinem Artikel „Container, die sich nie ändern“ im Dr. Dobbs Journal vom 19. September 2013 präsentierte Andrew Koenig ein interessantes Beispiel für algorithmische Ineffizienz, wenn ein Programmierstil verwendet wird, bei dem Variablen nach der Initialisierung unveränderlich sind. Bei diesem Stil werden Schleifen im Allgemeinen mit Rekursion ausgedrückt. Bei einigen Algorithmen, z. B. beim Generieren einer Collatz-Sequenz, erfordert die Rekursion das logische Kopieren eines Containers:

```
// Based on an example by Andrew Koenig in his Dr. Dobbs Journal article
```

```
// “Containers That Never Change” September 19, 2013, available at
// <url: http://www.drdobbs.com/cpp/containters-that-never-change/240161543>

// Includes here, e.g. <vector>

namespace my {
    template< class Item >
    using Vector_ = /* E.g. std::vector<Item> */;

    auto concat( Vector_<int> const& v, int const x )
        -> Vector_<int>
    {
        auto result{ v };
        result.push_back( x );
        return result;
    }

    auto collatz_aux( int const n, Vector_<int> const& result )
        -> Vector_<int>
    {
        if( n == 1 )
        {
            return result;
        }
        auto const new_result = concat( result, n );
        if( n % 2 == 0 )
        {
            return collatz_aux( n/2, new_result );
        }
        else
        {
            return collatz_aux( 3*n + 1, new_result );
        }
    }

    auto collatz( int const n )
        -> Vector_<int>
    {
        assert( n != 0 );
        return collatz_aux( n, Vector_<int>() );
    }
}  // namespace my

#include <iostream>
using namespace std;
auto main() -> int
{
    for( int const x : my::collatz( 42 ) )
    {
        cout << x << ' ';
    }
    cout << '\n';
}
```

Ausgabe:

```
42 21 64 32 16 8 4 2
```

Die Anzahl der Elementkopiervorgänge aufgrund des Kopierens der Vektoren beträgt hier

ungefähr 0 (*n²*), da es sich um die Summe 1 + 2 + 3 + ... *n handelt* .

In konkreten Zahlen führte bei g ++ - und Visual C ++ - Compilern der obige Aufruf von `collatz(42)` zu einer Collatz-Sequenz von 8 Elementen und 36 `collatz(42)` 8 * `collatz(42)` = 28, plus einige) in Vektorkopiekonstruktoraufrufen.

Alle diese Elementkopiervorgänge können entfernt werden, indem Sie einfach Vektoren verschieben, deren Werte nicht mehr benötigt werden. Dazu müssen Sie `const` und reference für die Vektortypargumente entfernen und die Vektoren *nach Wert übergeben* . Die Funktionsrückgaben werden bereits automatisch optimiert. Für die Aufrufe, bei denen Vektoren übergeben und nicht weiter verwendet werden, wenden Sie einfach `std::move` an, *um* diese Puffer zu *verschieben* , anstatt sie tatsächlich zu kopieren:

```
using std::move;

auto concat( Vector_<int> v, int const x )
    -> Vector_<int>
{
    v.push_back( x );
    // warning: moving a local object in a return statement prevents copy elision [-
Wpessimizing-move]
    // See https://stackoverflow.com/documentation/c%2b%2b/2489/copy-elision
    // return move( v );
    return v;
}

auto collatz_aux( int const n, Vector_<int> result )
    -> Vector_<int>
{
    if( n == 1 )
    {
        return result;
    }
    auto new_result = concat( move( result ), n );
    struct result;      // Make absolutely sure no use of `result` after this.
    if( n % 2 == 0 )
    {
        return collatz_aux( n/2, move( new_result ) );
    }
    else
    {
        return collatz_aux( 3*n + 1, move( new_result ) );
    }
}

auto collatz( int const n )
    -> Vector_<int>
{
    assert( n != 0 );
    return collatz_aux( n, Vector_<int>() );
}
```

Bei g ++ - und Visual C ++ - Compilern betrug die Anzahl der Elementkopiervorgänge aufgrund von Vektorkopiekonstruktoraufrufen genau 0.

Der Algorithmus ist notwendigerweise immer noch O *(n)* in der Länge der Collatz - Sequenz

erzeugt, aber das ist eine ziemlich dramatische Verbesserung: O *(n* ²) → O *(n).*

Mit etwas Sprachunterstützung könnte man vielleicht Moving verwenden und dennoch die Unveränderlichkeit einer Variablen *zwischen ihrer Initialisierung und ihrer endgültigen Verschiebung* ausdrücken und durchsetzen, wonach jede Verwendung dieser Variablen ein Fehler sein sollte. Leider unterstützt C ++ ab C ++ 14 das nicht. Bei schleifenfreiem Code kann die Nichtbenutzung nach dem Verschieben durch eine erneute Deklaration des betreffenden Namens als unvollständige `struct` erzwungen werden, wie bei `struct result;` oben, aber das ist hässlich und wird von anderen Programmierern wahrscheinlich nicht verstanden; Auch die Diagnose kann irreführend sein.

Zusammenfassend lässt sich sagen, dass die Unterstützung für das Verschieben von C ++ - Sprache und -Bibliothek drastische Verbesserungen der Algorithmuskomplexität ermöglicht, jedoch aufgrund der Unvollständigkeit der Unterstützung auf Kosten der Verzicht auf die Garantien für die Korrektheit des Codes und die Klarheit, die `const` sich bringt.

Der Vollständigkeit halber wird die instrumentierte Vektorklasse zum Messen der Anzahl der Elementkopiervorgänge aufgrund von Konstruktoraufrufen verwendet:

```
template< class Item >
class Copy_tracking_vector
{
private:
    static auto n_copy_ops()
        -> int&
    {
        static int value;
        return value;
    }

    vector<Item>    items_;

public:
    static auto n() -> int { return n_copy_ops(); }

    void push_back( Item const& o ) { items_.push_back( o ); }
    auto begin() const { return items_.begin(); }
    auto end() const { return items_.end(); }

    Copy_tracking_vector(){}

    Copy_tracking_vector( Copy_tracking_vector const& other )
        : items_( other.items_ )
    { n_copy_ops() += items_.size(); }

    Copy_tracking_vector( Copy_tracking_vector&& other )
        : items_( move( other.items_ ) )
    {}
};
```

Verwenden der Verschiebesemantik für Container

Sie können einen Container verschieben, anstatt ihn zu kopieren:

```
void print(const std::vector<int>& vec) {
    for (auto&& val : vec) {
        std::cout << val << ", ";
    }
    std::cout << std::endl;
}

int main() {
    // initialize vec1 with 1, 2, 3, 4 and vec2 as an empty vector
    std::vector<int> vec1{1, 2, 3, 4};
    std::vector<int> vec2;

    // The following line will print 1, 2, 3, 4
    print(vec1);

    // The following line will print a new line
    print(vec2);

    // The vector vec2 is assigned with move assingment.
    // This will "steal" the value of vec1 without copying it.
    vec2 = std::move(vec1);

    // Here the vec1 object is in an indeterminate state, but still valid.
    // The object vec1 is not destroyed,
    // but there's is no guarantees about what it contains.

    // The following line will print 1, 2, 3, 4
    print(vec2);
}
```

Verwenden Sie ein verschobenes Objekt erneut

Sie können ein verschobenes Objekt wiederverwenden:

```
void consumingFunction(std::vector<int> vec) {
    // Some operations
}

int main() {
    // initialize vec with 1, 2, 3, 4
    std::vector<int> vec{1, 2, 3, 4};

    // Send the vector by move
    consumingFunction(std::move(vec));

    // Here the vec object is in an indeterminate state.
    // Since the object is not destroyed, we can assign it a new content.
    // We will, in this case, assign an empty value to the vector,
    // making it effectively empty
    vec = {};

    // Since the vector as gained a determinate value, we can use it normally.
    vec.push_back(42);

    // Send the vector by move again.
    consumingFunction(std::move(vec));
}
```

Semantik verschieben online lesen: https://riptutorial.com/de/cplusplus/topic/2129/semantik-verschieben

Kapitel 102: Semaphor

Einführung

Semaphore sind derzeit nicht in C ++ verfügbar, können jedoch leicht mit einem Mutex und einer Bedingungsvariablen implementiert werden.

Dieses Beispiel wurde entnommen aus:

C ++ 0x hat keine Semaphore? Wie synchronisiere ich Threads?

Examples

Semaphor C ++ 11

```
#include <mutex>
#include <condition_variable>

class Semaphore {
public:
    Semaphore (int count_ = 0)
    : count(count_)
    {
    }

    inline void notify( int tid ) {
        std::unique_lock<std::mutex> lock(mtx);
        count++;
        cout << "thread " << tid <<  " notify" << endl;
        //notify the waiting thread
        cv.notify_one();
    }
    inline void wait( int tid ) {
        std::unique_lock<std::mutex> lock(mtx);
        while(count == 0) {
            cout << "thread " << tid << " wait" << endl;
            //wait on the mutex until notify is called
            cv.wait(lock);
            cout << "thread " << tid << " run" << endl;
        }
        count--;
    }
private:
    std::mutex mtx;
    std::condition_variable cv;
    int count;
};
```

Semaphore-Klasse in Aktion

Die folgende Funktion fügt vier Threads hinzu. Drei Threads konkurrieren um das Semaphor, das auf eine Anzahl von Eins gesetzt ist. Ein langsamerer Thread ruft `notify_one()` , damit einer der

wartenden Threads fortfahren kann.

Das Ergebnis ist , dass `s1` sofort beginnt sich zu drehen, wodurch die Nutzung des Semaphore `count` unter 1. Die anderen Fäden wiederum von der Bedingungsvariable warten zu bleiben , bis benachrichtigen () aufgerufen wird.

```
int main()
{
    Semaphore sem(1);

    thread s1([&]() {
        while(true) {
            this_thread::sleep_for(std::chrono::seconds(5));
            sem.wait( 1 );
        }
    });
    thread s2([&]() {
        while(true){
            sem.wait( 2 );
        }
    });
    thread s3([&]() {
        while(true) {
            this_thread::sleep_for(std::chrono::milliseconds(600));
            sem.wait( 3 );
        }
    });
    thread s4([&]() {
        while(true) {
            this_thread::sleep_for(std::chrono::seconds(5));
            sem.notify( 4 );
        }
    });


    s1.join();
    s2.join();
    s3.join();
    s4.join();

    ...
    }
```

Semaphor online lesen: https://riptutorial.com/de/cplusplus/topic/9785/semaphor

Kapitel 103: SFINAE (Substitutionsfehler ist kein Fehler)

Examples

enable_if

`std::enable_if` ist ein praktisches Dienstprogramm, um boolesche Bedingungen zum Auslösen von SFINAE zu verwenden. Es ist definiert als:

```
template <bool Cond, typename Result=void>
struct enable_if { };

template <typename Result>
struct enable_if<true, Result> {
    using type = Result;
};
```

Das heißt, `enable_if<true, R>::type` ist ein Alias für `R` , während `enable_if<false, T>::type` wird als die Spezialisierung der schlecht gebildeten `enable_if` keinen haben `type` Elementtyp.

`std::enable_if` können Sie Vorlagen einschränken:

```
int negate(int i) { return -i; }

template <class F>
auto negate(F f) { return -f(); }
```

Hier würde ein Aufruf zum `negate(1)` aufgrund von Mehrdeutigkeit fehlschlagen. Die zweite Überladung soll jedoch nicht für ganzzahlige Typen verwendet werden. Daher können wir Folgendes hinzufügen:

```
int negate(int i) { return -i; }

template <class F, class = typename std::enable_if<!std::is_arithmetic<F>::value>::type>
auto negate(F f) { return -f(); }
```

Jetzt Instanziieren `negate<int>` würde , da in einem Substitutionsfehler führen `!std::is_arithmetic<int>::value` ist `false` . Aufgrund von SFINAE ist dies kein schwerwiegender Fehler. Dieser Kandidat wird einfach aus dem Überlastungssatz entfernt. `negate(1)` nur einen einzigen möglichen Kandidaten - der dann aufgerufen wird.

Wann verwenden?

Es ist erwähnenswert, dass `std::enable_if` *auf* `std::enable_if` ein Helfer ist, aber es ist nicht der Grund, warum SFINAE überhaupt erst funktioniert. Wir betrachten diese beiden Alternativen für

die Implementierung einer Funktionalität, die der von `std::size` ähnelt, dh einer Überladungssatzgröße `size(arg)` , die die Größe eines Containers oder Arrays erzeugt:

```
// for containers
template<typename Cont>
auto size1(Cont const& cont) -> decltype( cont.size() );

// for arrays
template<typename Elt, std::size_t Size>
std::size_t size1(Elt const(&arr)[Size]);

// implementation omitted
template<typename Cont>
struct is_sizeable;

// for containers
template<typename Cont, std::enable_if_t<std::is_sizeable<Cont>::value, int> = 0>
auto size2(Cont const& cont);

// for arrays
template<typename Elt, std::size_t Size>
std::size_t size2(Elt const(&arr)[Size]);
```

Unter der Annahme, dass `is_sizeable` entsprechend geschrieben ist, sollten diese beiden Deklarationen in Bezug auf SFINAE genau gleichwertig sein. Welches ist am einfachsten zu schreiben und welches ist am einfachsten auf einen Blick zu überprüfen und zu verstehen?

Lassen Sie uns nun überlegen, wie wir möglicherweise arithmetische Helfer implementieren möchten, die den Überlauf von signierten Integerzahlen zugunsten von Wrap-around oder modularem Verhalten vermeiden. Das heißt, dass zum Beispiel `incr(i, 3)` das gleiche wie `i += 3` wäre, `INT_MAX` , dass das Ergebnis immer definiert würde, selbst wenn `i` ein `int` mit dem Wert `INT_MAX` . Dies sind zwei mögliche Alternativen:

```
// handle signed types
template<typename Int>
auto incr1(Int& target, Int amount)
-> std::void_t<int[static_cast<Int>(-1) < static_cast<Int>(0)]>;

// handle unsigned types by just doing target += amount
// since unsigned arithmetic already behaves as intended
template<typename Int>
auto incr1(Int& target, Int amount)
-> std::void_t<int[static_cast<Int>(0) < static_cast<Int>(-1)]>;

template<typename Int, std::enable_if_t<std::is_signed<Int>::value, int> = 0>
void incr2(Int& target, Int amount);

template<typename Int, std::enable_if_t<std::is_unsigned<Int>::value, int> = 0>
void incr2(Int& target, Int amount);
```

Welches ist am einfachsten zu schreiben und welches ist am einfachsten auf einen Blick zu überprüfen und zu verstehen?

Eine Stärke von `std::enable_if` ist, wie es mit Refactoring und API-Design spielt. Wenn `is_sizeable<Cont>::value` bedeuten soll, ob `cont.size()` gültig ist, kann die Verwendung des

Ausdrucks, wie er für `size1` erscheint, `size1` sein, obwohl dies davon abhängen könnte, ob `is_sizeable` an mehreren Stellen verwendet wird oder nicht . `std::is_signed` das mit `std::is_signed` was seine Absicht viel deutlicher widerspiegelt, als wenn die Implementierung in die Deklaration von `incr1` .

void_t

C ++ 11

`void_t` ist eine Meta-Funktion, die beliebige (Anzahl) Typen dem Typ `void` `void_t` . Der Hauptzweck von `void_t` ist es, das Schreiben von `void_t` zu erleichtern.

`std::void_t` wird Teil von C ++ 17 sein, aber bis dahin ist es äußerst einfach zu implementieren:

```
template <class...> using void_t = void;
```

Einige Compiler erfordern eine etwas andere Implementierung:

```
template <class...>
struct make_void { using type = void; };

template <typename... T>
using void_t = typename make_void<T...>::type;
```

Die Hauptanwendung von `void_t` ist das Schreiben von `void_t` , die die Gültigkeit einer Anweisung prüfen. Lassen Sie uns beispielsweise prüfen, ob ein Typ eine `foo()` , die keine Argumente `foo()` :

```
template <class T, class=void>
struct has_foo : std::false_type {};

template <class T>
struct has_foo<T, void_t<decltype(std::declval<T&>().foo())>> : std::true_type {};
```

Wie funktioniert das? Wenn ich versuche, `has_foo<T>::value` zu instantiieren, wird der Compiler versuchen, nach der besten Spezialisierung für `has_foo<T, void>` zu suchen. Wir haben zwei Optionen: die primäre und die sekundäre, bei der der zugrunde liegende Ausdruck instanziiert werden muss:

- Wenn `T` eine Member - Funktion *hat* `foo()` , dann gleich welcher Art, die wird konvertiert zurück `void` , und die Spezialisierung auf die primäre auf der Grundlage der Vorlage Teilordnungsregeln bevorzugt. Also ist `has_foo<T>::value` `true`
- Wenn `T` *über keine* solche Member-Funktion verfügt (oder mehr als ein Argument erforderlich ist), schlägt die Ersetzung für die Spezialisierung fehl und wir haben nur die primäre Vorlage, auf die zurückgegriffen werden kann. Daher ist `has_foo<T>::value` `false` .

Ein einfacherer Fall:

```
template<class T, class=void>
struct can_reference : std::false_type {};
```

```
template<class T>
struct can_reference<T, std::void_t<T&>> : std::true_type {};
```

dies verwendet nicht `std::declval` oder `decltype` .

Sie können ein allgemeines Muster eines ungültigen Arguments feststellen. Wir können das ausrechnen:

```
struct details {
  template<template<class...>class Z, class=void, class...Ts>
  struct can_apply:
    std::false_type
  {};
  template<template<class...>class Z, class...Ts>
  struct can_apply<Z, std::void_t<Z<Ts...>>, Ts...>:
    std::true_type
  {};
};

template<template<class...>class Z, class...Ts>
using can_apply = details::can_apply<Z, void, Ts...>;
```

`std::void_t` die Verwendung von `std::void_t` und `can_apply` als Indikator fungieren, der darauf `can_apply` , ob der als erstes Vorlagenargument bereitgestellte Typ nach dem Ersetzen der anderen Typen `can_apply` gebildet ist. Die vorherigen Beispiele können jetzt mit `can_apply` neu `can_apply` werden:

```
template<class T>
using ref_t = T&;

template<class T>
using can_reference = can_apply<ref_t, T>;    // Is T& well formed for T?
```

und:

```
template<class T>
using dot_foo_r = decltype(std::declval<T&>().foo());

template<class T>
using can_dot_foo = can_apply< dot_foo_r, T >;    // Is T.foo() well formed for T?
```

Das scheint einfacher als die Originalversionen.

Es gibt Post-C ++ 17 Vorschläge für `std` Eigenschaften ähnlich wie `can_apply` .

Der Nutzen von `void_t` wurde von Walter Brown entdeckt. Er hat auf der CppCon 2016 eine wunderbare Präsentation darüber gegeben.

nachfolgender decltype in Funktionsvorlagen

C ++ 11

Eine der einschränkenden Funktionen besteht darin, den Rückgabetyp mit einem `decltype` anzugeben:

```
namespace details {
   using std::to_string;

   // this one is constrained on being able to call to_string(T)
   template <class T>
   auto convert_to_string(T const& val, int )
       -> decltype(to_string(val))
   {
       return to_string(val);
   }

   // this one is unconstrained, but less preferred due to the ellipsis argument
   template <class T>
   std::string convert_to_string(T const& val, ... )
   {
       std::ostringstream oss;
       oss << val;
       return oss.str();
   }
}

template <class T>
std::string convert_to_string(T const& val)
{
    return details::convert_to_string(val, 0);
}
```

Wenn ich `convert_to_string()` mit einem Argument aufrufen, mit dem ich `to_string()` aufrufen kann, `to_string()` ich zwei `details::convert_to_string()` Funktionen für `details::convert_to_string()` . Die erste ist bevorzugt, da die Umwandlung von `0` in `int` eine bessere implizite Umwandlungssequenz ist als die Umwandlung von `0` in `...`

Wenn ich `convert_to_string()` mit einem Argument aufrufen, aus dem ich `to_string()` nicht aufrufen `to_string()` , führt die Instanziierung der ersten Funktionsschablone zum Substitutionsfehler (es gibt keinen `decltype(to_string(val))`). Dieser Kandidat wird daher aus dem Überlastungssatz entfernt. Die zweite Funktionsvorlage ist nicht eingeschränkt, also wird sie ausgewählt und wir gehen stattdessen durch den `operator<<(std::ostream&, T)` . Wenn dieser nicht definiert ist, liegt ein schwerwiegender Kompilierungsfehler mit einem Vorlagenstapel in der Zeile `oss << val` .

Was ist SFINAE?

SFINAE steht für **S** ubstitution **F** ailure **I** s **N** **A** n **E** rror ot. Ungeformter Code, der sich aus dem Ersetzen von Typen (oder Werten) ergibt, um eine Funktionsvorlage oder eine Klassenvorlage zu instanziieren, ist **kein** schwerwiegender Kompilierungsfehler, sondern wird nur als Deduktionsfehler behandelt.

Abzugsfehler bei der Instanziierung von Funktionsvorlagen oder Klassenvorlagen-Spezialisierungen entfernen diesen Kandidaten aus der Menge der Gegenleistung - als ob der fehlerhafte Kandidat nicht vorhanden wäre.

```
template <class T>
auto begin(T& c) -> decltype(c.begin()) { return c.begin(); }

template <class T, size_t N>
T* begin(T (&arr)[N]) { return arr; }

int vals[10];
begin(vals); // OK. The first function template substitution fails because
             // vals.begin() is ill-formed. This is not an error! That function
             // is just removed from consideration as a viable overload candidate,
             // leaving us with the array overload.
```

Nur Substitutionsfehler im **unmittelbaren Kontext** werden als Abzugsfehler betrachtet, alle anderen werden als harte Fehler betrachtet.

```
template <class T>
void add_one(T& val) { val += 1; }

int i = 4;
add_one(i); // ok

std::string msg = "Hello";
add_one(msg); // error. msg += 1 is ill-formed for std::string, but this
              // failure is NOT in the immediate context of substituting T
```

enable_if_all / enable_if_any

C ++ 11

Motivationsbeispiel

Wenn Sie ein variadisches Vorlagenpaket in der Vorlagenparameterliste haben, wie im folgenden Codeausschnitt:

```
template<typename ...Args> void func(Args &&...args) { //... };
```

Die Standardbibliothek (vor C ++ 17) bietet keine direkte Möglichkeit, **enable_if** zu schreiben, um SFINAE-Einschränkungen für ***alle Parameter*** in `Args` oder für ***alle Parameter*** in `Args` . C ++ 17 bietet `std::conjunction` und `std::disjunction` die dieses Problem lösen. Zum Beispiel:

```
/// C++17: SFINAE constraints on all of the parameters in Args.
template<typename ...Args,
         std::enable_if_t<std::conjunction_v<custom_conditions_v<Args>...>>* = nullptr>
void func(Args &&...args) { //... };

/// C++17: SFINAE constraints on any of the parameters in Args.
template<typename ...Args,
         std::enable_if_t<std::disjunction_v<custom_conditions_v<Args>...>>* = nullptr>
void func(Args &&...args) { //... };
```

Wenn Sie nicht über C ++ 17 verfügen, gibt es mehrere Lösungen, um dies zu erreichen. Eine davon besteht darin, eine Basisfallklasse und **Teilspezialisierungen zu verwenden** , wie die

Antworten auf diese Frage zeigen .

Alternativ kann man auch das Verhalten von `std::conjunction` und `std::disjunction` auf recht einfache Weise implementieren. Im folgenden Beispiel werde ich die Implementierungen demonstrieren und sie mit `std::enable_if` , um zwei Aliasnamen zu erzeugen: `enable_if_all` und `enable_if_any` , die genau das tun, was sie semantisch tun sollen. Dies kann eine skalierbarere Lösung bieten.

Implementierung von `enable_if_all` und `enable_if_any`

Zuerst emulieren Sie `std::conjunction` und `std::disjunction` Verwendung von benutzerdefiniertem `seq_and` bzw. `seq_or` :

```
/// Helper for prior to C++14.
template<bool B, class T, class F >
using conditional_t = typename std::conditional<B,T,F>::type;

/// Emulate C++17 std::conjunction.
template<bool...> struct seq_or: std::false_type {};
template<bool...> struct seq_and: std::true_type {};

template<bool B1, bool... Bs>
struct seq_or<B1,Bs...>:
  conditional_t<B1,std::true_type,seq_or<Bs...>> {};

template<bool B1, bool... Bs>
struct seq_and<B1,Bs...>:
  conditional_t<B1,seq_and<Bs...>,std::false_type> {};
```

Dann ist die Implementierung recht unkompliziert:

```
template<bool... Bs>
using enable_if_any = std::enable_if<seq_or<Bs...>::value>;

template<bool... Bs>
using enable_if_all = std::enable_if<seq_and<Bs...>::value>;
```

Eventuell einige Helfer:

```
template<bool... Bs>
using enable_if_any_t = typename enable_if_any<Bs...>::type;

template<bool... Bs>
using enable_if_all_t = typename enable_if_all<Bs...>::type;
```

Verwendungszweck

Die Verwendung ist auch unkompliziert:

```
    /// SFINAE constraints on all of the parameters in Args.
    template<typename ...Args,
             enable_if_all_t<custom_conditions_v<Args>...>* = nullptr>
    void func(Args &&...args) { //... };

    /// SFINAE constraints on any of the parameters in Args.
    template<typename ...Args,
             enable_if_any_t<custom_conditions_v<Args>...>* = nullptr>
    void func(Args &&...args) { //... };
```

ist angeschlossen

Um die Typ_trait-Erzeugung zu verallgemeinern: Basierend auf SFINAE gibt es experimentelle Merkmale, die `detected_or` , `detected_t` und `is_detected` .

Mit Template-Parametern `typename Default` , `template <typename...> Op` und `typename ... Args` :

- `is_detected` : Alias von `std::true_type` oder `std::false_type` abhängig von der Gültigkeit von `Op<Args...>`
- `detected_t` : Alias für `Op<Args...>` oder `nonesuch` in Abhängigkeit von ihrer Gültigkeit `Op<Args...>` .
- `detected_or` : alias eine Struktur mit `value_t` welcher `is_detected` und `type` das ist `Op<Args...>` oder `Default` in Abhängigkeit von ihrer Gültigkeit `Op<Args...>`

`std::void_t` kann mit `std::void_t` für SFINAE wie folgt implementiert werden:

C ++ 17

```
namespace detail {
    template <class Default, class AlwaysVoid,
              template<class...> class Op, class... Args>
    struct detector
    {
        using value_t = std::false_type;
        using type = Default;
    };

    template <class Default, template<class...> class Op, class... Args>
    struct detector<Default, std::void_t<Op<Args...>>, Op, Args...>
    {
        using value_t = std::true_type;
        using type = Op<Args...>;
    };

} // namespace detail

// special type to indicate detection failure
struct nonesuch {
    nonesuch() = delete;
    ~nonesuch() = delete;
    nonesuch(nonesuch const&) = delete;
    void operator=(nonesuch const&) = delete;
};

template <template<class...> class Op, class... Args>
using is_detected =
```

```
    typename detail::detector<nonesuch, void, Op, Args...>::value_t;

template <template<class...> class Op, class... Args>
using detected_t = typename detail::detector<nonesuch, void, Op, Args...>::type;

template <class Default, template<class...> class Op, class... Args>
using detected_or = detail::detector<Default, void, Op, Args...>;
```

Eigenschaften zum Erkennen des Vorhandenseins einer Methode können dann einfach implementiert werden:

```
typename <typename T, typename ...Ts>
using foo_type = decltype(std::declval<T>().foo(std::declval<Ts>()...));

struct C1 {};

struct C2 {
    int foo(char) const;
};

template <typename T>
using has_foo_char = is_detected<foo_type, T, char>;

static_assert(!has_foo_char<C1>::value, "Unexpected");
static_assert(has_foo_char<C2>::value, "Unexpected");

static_assert(std::is_same<int, detected_t<foo_type, C2, char>>::value,
              "Unexpected");

static_assert(std::is_same<void, // Default
                           detected_or<void, foo_type, C1, char>>::value,
              "Unexpected");
static_assert(std::is_same<int, detected_or<void, foo_type, C2, char>>::value,
              "Unexpected");
```

Überlastauflösung mit vielen Optionen

Wenn Sie zwischen mehreren Optionen auswählen müssen, kann die Aktivierung nur einer über `enable_if<>` ziemlich umständlich sein, da mehrere Bedingungen ebenfalls negiert werden müssen.

Die Reihenfolge zwischen Überladungen kann stattdessen über Vererbung, dh Tag-Versand, ausgewählt werden.

Anstatt zu testen, was gut `decltype` , und auch die Negation aller anderen Versionsbedingungen zu testen, testen wir stattdessen genau das, was wir brauchen, vorzugsweise in einem `decltype` in einer nachlaufenden Rückgabe.
Dies kann mehrere Optionen `random_access_tag` , wir unterscheiden zwischen denjenigen, die "Tags" verwenden, ähnlich wie bei Iterator-Trait-Tags (`random_access_tag` et al). Dies funktioniert, weil eine direkte Übereinstimmung besser ist als eine Basisklasse. Dies ist besser als eine Basisklasse einer Basisklasse usw.

```
#include <algorithm>
#include <iterator>
```

```
namespace detail
{
    // this gives us infinite types, that inherit from each other
    template<std::size_t N>
    struct pick : pick<N-1> {};
    template<>
    struct pick<0> {};

    // the overload we want to be preferred have a higher N in pick<N>
    // this is the first helper template function
    template<typename T>
    auto stable_sort(T& t, pick<2>)
        -> decltype( t.stable_sort(), void() )
    {
        // if the container have a member stable_sort, use that
        t.stable_sort();
    }

    // this helper will be second best match
    template<typename T>
    auto stable_sort(T& t, pick<1>)
        -> decltype( t.sort(), void() )
    {
        // if the container have a member sort, but no member stable_sort
        // it's customary that the sort member is stable
        t.sort();
    }

    // this helper will be picked last
    template<typename T>
    auto stable_sort(T& t, pick<0>)
        -> decltype( std::stable_sort(std::begin(t), std::end(t)), void() )
    {
        // the container have neither a member sort, nor member stable_sort
        std::stable_sort(std::begin(t), std::end(t));
    }

}

// this is the function the user calls. it will dispatch the call
// to the correct implementation with the help of 'tags'.
template<typename T>
void stable_sort(T& t)
{
    // use an N that is higher that any used above.
    // this will pick the highest overload that is well formed.
    detail::stable_sort(t, detail::pick<10>{});
}
```

Es gibt andere Methoden, die üblicherweise zur Unterscheidung zwischen Überladungen verwendet werden, z. B. dass exakte Übereinstimmungen besser sind als die Umwandlung und besser als Ellipsen.

Der Tag-Versand kann sich jedoch auf eine beliebige Anzahl von Optionen erstrecken und ist in der Absicht etwas klarer.

SFINAE (Substitutionsfehler ist kein Fehler) online lesen:
https://riptutorial.com/de/cplusplus/topic/1169/sfinae--substitutionsfehler-ist-kein-fehler-

Kapitel 104: Singleton Design Pattern

Bemerkungen

Ein **Singleton** soll sicherstellen, dass eine Klasse nur eine Instanz hat *und* einen globalen Zugriffspunkt bietet. Wenn Sie nur eine Instanz *oder* einen geeigneten globalen Zugriffspunkt benötigen, jedoch nicht beide, sollten Sie andere Optionen in Betracht ziehen, bevor Sie sich dem Einzelspieler zuwenden.

Globale Variablen *können* es schwieriger machen, über Code nachzudenken. Wenn zum Beispiel eine der aufrufenden Funktionen mit den Daten, die sie von einem Singleton empfängt, nicht zufrieden ist, müssen Sie jetzt herausfinden, was zuerst die fehlerhaften Daten des Singleton an erster Stelle gibt.

Singletons fördern auch die Kopplung , ein Begriff, der verwendet wird, um zwei Komponenten des Codes zu beschreiben, die miteinander verbunden sind, wodurch die eigenen Maßnahmen zur Selbsteinschließung jeder Komponente reduziert werden.

Singletons sind nicht gleichlaufsicher. Wenn eine Klasse über einen globalen Zugriffspunkt verfügt, kann jeder Thread darauf zugreifen, was zu Deadlocks und Race-Bedingungen führen kann.

Schließlich kann eine langsame Initialisierung zu Leistungsproblemen führen, wenn sie zum falschen Zeitpunkt initialisiert wird. Durch das Entfernen der verzögerten Initialisierung werden auch einige der Features entfernt, die Singletons erst recht interessant machen, wie etwa Polymorphismus (siehe Unterklassen).

Quellen: Spielprogrammierungsmuster von Robert Nystrom

Examples

Faule Initialisierung

Dieses Beispiel wurde aus dem `Q & A` Abschnitt hier entfernt:
http://stackoverflow.com/a/1008289/3807729

In diesem Artikel finden Sie ein einfaches Design für ein Lazy, das mit garantierter Zerstörung bewertet wird:
Kann mir jemand eine Probe von Singleton in c ++ zur Verfügung stellen?

Der klassische Lazy hat Singleton ausgewertet und richtig zerstört.

```
class S
{
    public:
        static S& getInstance()
        {
```

```
            static S    instance; // Guaranteed to be destroyed.
                                  // Instantiated on first use.
            return instance;
        }
    private:
        S() {};                   // Constructor? (the {} brackets) are needed here.

        // C++ 03
        // ========
        // Dont forget to declare these two. You want to make sure they
        // are unacceptable otherwise you may accidentally get copies of
        // your singleton appearing.
        S(S const&);              // Don't Implement
        void operator=(S const&); // Don't implement

        // C++ 11
        // =======
        // We can use the better technique of deleting the methods
        // we don't want.
    public:
        S(S const&)               = delete;
        void operator=(S const&)  = delete;

        // Note: Scott Meyers mentions in his Effective Modern
        //       C++ book, that deleted functions should generally
        //       be public as it results in better error messages
        //       due to the compilers behavior to check accessibility
        //       before deleted status
};
```

In diesem Artikel wird beschrieben, wann ein Singleton verwendet wird: (nicht oft)
Singleton: Wie soll es verwendet werden?

In diesen beiden Artikeln finden Sie Informationen zur Initialisierungsreihenfolge und zum Umgang damit:
Reihenfolge der Initialisierung der statischen Variablen
Probleme mit der statischen C ++ - Initialisierungsreihenfolge finden

In diesem Artikel wird die Lebensdauer beschrieben:
Wie lang ist eine statische Variable in einer C ++ - Funktion?

In diesem Artikel werden einige Threading-Implikationen für Singletons beschrieben:
Singleton-Instanz, die als statische Variable der GetInstance-Methode deklariert ist

In diesem Artikel wird erläutert, warum doppelt gesichertes Sperren in C ++ nicht funktioniert:
Was sind die allgemeinen undefinierten Verhaltensweisen, über die ein C ++ - Programmierer Bescheid wissen sollte?

Unterklassen

```
class API
{
public:
    static API& instance();
```

```
    virtual ~API() {}

    virtual const char* func1() = 0;
    virtual void func2() = 0;

protected:
    API() {}
    API(const API&) = delete;
    API& operator=(const API&) = delete;
};

class WindowsAPI : public API
{
public:
    virtual const char* func1()  override { /* Windows code */ }
    virtual void func2() override { /* Windows code */ }
};

class LinuxAPI : public API
{
public:
    virtual const char* func1() override { /* Linux code */ }
    virtual void func2() override { /* Linux code */ }
};

API& API::instance() {
#if PLATFORM == WIN32
    static WindowsAPI instance;
#elif PLATFORM = LINUX
    static LinuxAPI instance;
#endif
    return instance;
}
```

In diesem Beispiel bindet ein einfacher Compiler-Switch die `API` Klasse an die entsprechende Unterklasse. Auf diese Weise kann auf die `API` zugegriffen werden, ohne an plattformspezifischen Code gekoppelt zu sein.

Fadensicheres Singeton

C ++ 11

Die C ++ 11-Standards garantieren, dass die Initialisierung von Funktionsumfangsobjekten synchronisiert initialisiert wird. Dies kann verwendet werden, um einen Thread-sicheren Singleton mit verzögerter Initialisierung zu implementieren.

```
class Foo
{
public:
    static Foo& instance()
    {
        static Foo inst;
        return inst;
    }
private:
    Foo() {}
    Foo(const Foo&) = delete;
```

```
    Foo& operator =(const Foo&) = delete;
};
```

Statischer Deinitialisierungssicherer Singleton.

Es gibt Zeiten mit mehreren statischen Objekten, bei denen Sie sicherstellen müssen, dass der *Singleton* nicht zerstört wird, bis alle statischen Objekte, die den *Singleton verwenden, ihn* nicht mehr benötigen.

In diesem Fall kann `std::shared_ptr` verwendet werden, um den *Singleton* für alle Benutzer am Leben zu erhalten, selbst wenn die statischen Destruktoren am Ende des Programms aufgerufen werden:

```
class Singleton
{
public:
    Singleton(Singleton const&) = delete;
    Singleton& operator=(Singleton const&) = delete;

    static std::shared_ptr<Singleton> instance()
    {
        static std::shared_ptr<Singleton> s{new Singleton};
        return s;
    }

private:
    Singleton() {}
};
```

HINWEIS: Dieses Beispiel wird hier als Antwort im Abschnitt "Fragen und Antworten" angezeigt.

Singleton Design Pattern online lesen: https://riptutorial.com/de/cplusplus/topic/2713/singleton-design-pattern

Kapitel 105: Sortierung

Bemerkungen

Die Funktionsfamilie `std::sort` befindet sich in der `algorithm` .

Examples

Sortierreihenfolge Container mit vorgegebener Reihenfolge

Wenn die Werte in einem Container bestimmte Operatoren bereits überladen haben, kann `std::sort` mit spezialisierten Funktionen verwendet werden, um entweder aufsteigend oder absteigend zu sortieren:

C ++ 11

```
#include <vector>
#include <algorithm>
#include <functional>

std::vector<int> v = {5,1,2,4,3};

//sort in ascending order (1,2,3,4,5)
std::sort(v.begin(), v.end(), std::less<int>());

// Or just:
std::sort(v.begin(), v.end());

//sort in descending order (5,4,3,2,1)
std::sort(v.begin(), v.end(), std::greater<int>());

//Or just:
std::sort(v.rbegin(), v.rend());
```

C ++ 14

In C ++ 14 müssen wir nicht das Vorlagenargument für die Vergleichsfunktionsobjekte angeben, sondern lassen das Objekt auf der Grundlage dessen ableiten, worauf es übergeben wird:

```
std::sort(v.begin(), v.end(), std::less<>());     // ascending order
std::sort(v.begin(), v.end(), std::greater<>());  // descending order
```

Sortierreihenfolge-Container durch überladenen Operator weniger

Wenn keine Sortierfunktion übergeben wird, ordnet `std::sort` die Elemente durch Aufruf des `operator<` in Elementpaaren an, der einen Typ zurückgeben muss, der in `bool` (oder einfach in `bool`) konvertierbar ist. Grundtypen (Ganzzahlen, Gleitkommazahlen, Zeiger usw.) haben bereits Vergleichsoperatoren eingebaut.

Wir können diesen Operator überlasten die Standardeinstellung zu machen `sort` Anruf Arbeit auf benutzerdefinierte Typen.

```
// Include sequence containers
#include <vector>
#include <deque>
#include <list>

// Insert sorting algorithm
#include <algorithm>

class Base {
 public:

    // Constructor that set variable to the value of v
    Base(int v): variable(v) {
    }

    // Use variable to provide total order operator less
    //`this` always represents the left-hand side of the compare.
    bool operator<(const Base &b) const {
        return this->variable < b.variable;
    }

    int variable;
};

int main() {
    std::vector <Base> vector;
    std::deque <Base> deque;
    std::list <Base> list;

    // Create 2 elements to sort
    Base a(10);
    Base b(5);

    // Insert them into backs of containers
    vector.push_back(a);
    vector.push_back(b);

    deque.push_back(a);
    deque.push_back(b);

    list.push_back(a);
    list.push_back(b);

    // Now sort data using operator<(const Base &b) function
    std::sort(vector.begin(), vector.end());
    std::sort(deque.begin(), deque.end());
    // List must be sorted differently due to its design
    list.sort();

    return 0;
}
```

Sequenzcontainer mit der Compare-Funktion sortieren

```
// Include sequence containers
```

```
#include <vector>
#include <deque>
#include <list>

// Insert sorting algorithm
#include <algorithm>

class Base {
 public:

    // Constructor that set variable to the value of v
    Base(int v): variable(v) {
    }

    int variable;
};

bool compare(const Base &a, const Base &b) {
    return a.variable < b.variable;
}

int main() {
    std::vector <Base> vector;
    std::deque <Base> deque;
    std::list <Base> list;

    // Create 2 elements to sort
    Base a(10);
    Base b(5);

    // Insert them into backs of containers
    vector.push_back(a);
    vector.push_back(b);

    deque.push_back(a);
    deque.push_back(b);

    list.push_back(a);
    list.push_back(b);

    // Now sort data using comparing function
    std::sort(vector.begin(), vector.end(), compare);
    std::sort(deque.begin(), deque.end(), compare);
    list.sort(compare);

    return 0;
}
```

Sequenzcontainer mit Lambda-Ausdrücken sortieren (C ++ 11)

C ++ 11

```
// Include sequence containers
#include <vector>
#include <deque>
#include <list>
#include <array>
#include <forward_list>
```

```
// Include sorting algorithm
#include <algorithm>

class Base {
 public:

    // Constructor that set variable to the value of v
    Base(int v): variable(v) {
    }

    int variable;
};


int main() {
    // Create 2 elements to sort
    Base a(10);
    Base b(5);

    // We're using C++11, so let's use initializer lists to insert items.
    std::vector <Base> vector = {a, b};
    std::deque <Base> deque = {a, b};
    std::list <Base> list = {a, b};
    std::array <Base, 2> array = {a, b};
    std::forward_list<Base> flist = {a, b};

    // We can sort data using an inline lambda expression
    std::sort(std::begin(vector), std::end(vector),
      [](const Base &a, const Base &b) { return a.variable < b.variable;});

    // We can also pass a lambda object as the comparator
    // and reuse the lambda multiple times
    auto compare = [](const Base &a, const Base &b) {
                    return a.variable < b.variable;};
    std::sort(std::begin(deque), std::end(deque), compare);
    std::sort(std::begin(array), std::end(array), compare);
    list.sort(compare);
    flist.sort(compare);

    return 0;
}
```

Container sortieren und sortieren

`std::sort` , im Standard-Bibliotheks-Header- `algorithm` , ist ein Standard-Bibliotheksalgorithmus zum Sortieren eines Wertebereichs, der von einem Iteratorpaar definiert wird. `std::sort` wird als letzter Parameter ein Functor verwendet, mit dem zwei Werte verglichen werden. So bestimmt es die Reihenfolge. Beachten Sie, dass `std::sort` nicht stabil ist .

Die Vergleichsfunktion *muss* den Elementen eine strikte, schwache Anordnung auferlegen. Ein einfacher Vergleich (oder mehr als) ist ausreichend.

Ein Container mit Iteratoren mit wahlfreiem Zugriff kann mit dem `std::sort` Algorithmus `std::sort` werden:

C ++ 11

```
#include <vector>
#include <algorithm>

std::vector<int> MyVector = {3, 1, 2}

//Default comparison of <
std::sort(MyVector.begin(), MyVector.end());
```

`std::sort` müssen die Iteratoren Random-Access-Iteratoren sein. Die `std::forward_list` `std::list` und `std::forward_list` (für die C ++ 11 erforderlich ist) bieten keine Iteratoren mit wahlfreiem Zugriff, daher können sie nicht mit `std::sort` . Sie verfügen jedoch über `sort` , die einen Sortieralgorithmus implementieren, der mit ihren eigenen Iteratortypen arbeitet.

C ++ 11

```
#include <list>
#include <algorithm>

std::list<int> MyList = {3, 1, 2}

//Default comparison of <
//Whole list only.
MyList.sort();
```

Ihre Mitglieder- `sort` immer die gesamte Liste, sodass sie keinen Teilbereich von Elementen sortieren können. Da `list` und `forward_list` jedoch schnelle Verknüpfungsoperationen haben, können Sie die zu sortierenden Elemente aus der Liste extrahieren, sortieren und dann dorthin zurückstellen, wo sie ziemlich effizient waren:

```
void sort_sublist(std::list<int>& mylist, std::list<int>::const_iterator start,
std::list<int>::const_iterator end) {
    //extract and sort half-open sub range denoted by start and end iterator
    std::list<int> tmp;
    tmp.splice(tmp.begin(), list, start, end);
    tmp.sort();
    //re-insert range at the point we extracted it from
    list.splice(end, tmp);
}
```

Sortierung mit std :: map (aufsteigend und absteigend)

In diesem Beispiel werden Elemente in **aufsteigender** Reihenfolge eines **Schlüssels** anhand einer Karte sortiert. Sie können im folgenden Beispiel anstelle von `std::string` einen beliebigen Typ verwenden, einschließlich class.

```
#include <iostream>
#include <utility>
#include <map>

int main()
{
    std::map<double, std::string> sorted_map;
    // Sort the names of the planets according to their size
    sorted_map.insert(std::make_pair(0.3829, "Mercury"));
```

```
    sorted_map.insert(std::make_pair(0.9499, "Venus"));
    sorted_map.insert(std::make_pair(1,      "Earth"));
    sorted_map.insert(std::make_pair(0.532,  "Mars"));
    sorted_map.insert(std::make_pair(10.97,  "Jupiter"));
    sorted_map.insert(std::make_pair(9.14,   "Saturn"));
    sorted_map.insert(std::make_pair(3.981,  "Uranus"));
    sorted_map.insert(std::make_pair(3.865,  "Neptune"));

    for (auto const& entry: sorted_map)
    {
        std::cout << entry.second << " (" << entry.first << " of Earth's radius)" << '\n';
    }
}
```

Ausgabe:

```
Mercury (0.3829 of Earth's radius)
Mars (0.532 of Earth's radius)
Venus (0.9499 of Earth's radius)
Earth (1 of Earth's radius)
Neptune (3.865 of Earth's radius)
Uranus (3.981 of Earth's radius)
Saturn (9.14 of Earth's radius)
Jupiter (10.97 of Earth's radius)
```

Wenn Einträge mit gleichen Schlüsseln möglich sind, verwenden Sie `multimap` anstelle von `map` (wie im folgenden Beispiel).

Um Elemente **absteigend** zu sortieren, deklarieren Sie die Karte mit einem entsprechenden Vergleichsfunktionscode (`std::greater<>`):

```
#include <iostream>
#include <utility>
#include <map>

int main()
{
    std::multimap<int, std::string, std::greater<int>> sorted_map;
    // Sort the names of animals in descending order of the number of legs
    sorted_map.insert(std::make_pair(6,   "bug"));
    sorted_map.insert(std::make_pair(4,   "cat"));
    sorted_map.insert(std::make_pair(100, "centipede"));
    sorted_map.insert(std::make_pair(2,   "chicken"));
    sorted_map.insert(std::make_pair(0,   "fish"));
    sorted_map.insert(std::make_pair(4,   "horse"));
    sorted_map.insert(std::make_pair(8,   "spider"));

    for (auto const& entry: sorted_map)
    {
        std::cout << entry.second << " (has " << entry.first << " legs)" << '\n';
    }
}
```

Ausgabe

```
centipede (has 100 legs)
```

```
spider (has 8 legs)
bug (has 6 legs)
cat (has 4 legs)
horse (has 4 legs)
chicken (has 2 legs)
fish (has 0 legs)
```

Eingebaute Arrays sortieren

Der `sort` sortiert eine Sequenz von zwei Iteratoren definiert. Dies reicht aus, um ein eingebautes Array (auch als c-Stil bezeichnet) zu sortieren.

C ++ 11

```
int arr1[] = {36, 24, 42, 60, 59};

// sort numbers in ascending order
sort(std::begin(arr1), std::end(arr1));

// sort numbers in descending order
sort(std::begin(arr1), std::end(arr1), std::greater<int>());
```

Vor C ++ 11 musste das Ende des Arrays anhand der Größe des Arrays "berechnet" werden:

C ++ 11

```
// Use a hard-coded number for array size
sort(arr1, arr1 + 5);

// Alternatively, use an expression
const size_t arr1_size = sizeof(arr1) / sizeof(*arr1);
sort(arr1, arr1 + arr1_size);
```

Sortierung online lesen: https://riptutorial.com/de/cplusplus/topic/1675/sortierung

Kapitel 106: Speicherklassenspezifizierer

Einführung

Speicherklassenspezifizierer sind Schlüsselwörter , die in Deklarationen verwendet werden können. Sie wirken sich nicht auf den Typ der Deklaration aus, ändern jedoch normalerweise die Art und Weise, in der die Entität gespeichert wird.

Bemerkungen

Es gibt sechs Speicherklassenspezifizierer, allerdings nicht alle in derselben Sprachversion: `auto` (bis C ++ 11), `register` (bis C ++ 17), `static` , `thread_local` (seit C ++ 11), `extern` und `mutable` .

Nach dem Standard

> Es darf höchstens ein *Speicherklassenspezifizierer* in einem angegebenen `thread_local` *-seq vorkommen, mit der* Ausnahme, dass `thread_local static` oder `extern` .

Eine Deklaration darf keinen Speicherklassenspezifizierer enthalten. In diesem Fall gibt die Sprache ein Standardverhalten an. Standardmäßig hat eine im Blockbereich deklarierte Variable implizit eine automatische Speicherdauer.

Examples

veränderlich

Ein Bezeichner, der auf die Deklaration eines nicht statischen Nicht-Referenzdatenelements einer Klasse angewendet werden kann. Ein veränderlicher Member einer Klasse ist nicht `const` selbst wenn das Objekt `const` .

```
class C {
    int x;
    mutable int times_accessed;
  public:
    C(): x(0), times_accessed(0) {
    }
    int get_x() const {
        ++times_accessed; // ok: const member function can modify mutable data member
        return x;
    }
    void set_x(int x) {
        ++times_accessed;
        this->x = x;
    }
};
```

C ++ 11

Eine zweite Bedeutung für `mutable` wurde in C ++ 11 hinzugefügt. Wenn es der Parameterliste eines Lambda folgt, unterdrückt es die implizite `const` Anweisung des Funktionsaufrufoperators des Lambda. Daher kann ein veränderliches Lambda die Werte von durch Kopie erfassten Entitäten ändern. Weitere Informationen finden Sie in den veränderbaren Lambdas .

```
std::vector<int> my_iota(int start, int count) {
    std::vector<int> result(count);
    std::generate(result.begin(), result.end(),
                  [start]() mutable { return start++; });
    return result;
}
```

Beachten Sie, dass `mutable` *kein* Speicherklassenspezifizierer ist, wenn auf diese Weise ein mutierbares Lambda gebildet wird.

registrieren

C ++ 17

Ein Speicherklassenspezifizierer, der den Compiler darauf hinweist, dass eine Variable stark verwendet wird. Das Wort "Register" bezieht sich auf die Tatsache, dass ein Compiler eine solche Variable in einem CPU-Register speichern kann, so dass in weniger Taktzyklen darauf zugegriffen werden kann. Es wurde ab C ++ 11 veraltet.

```
register int i = 0;
while (i < 100) {
    f(i);
    int g = i*i;
    i += h(i, g);
}
```

Sowohl lokale Variablen als auch Funktionsparameter können als `register` deklariert `register` . Im Gegensatz zu C gibt C ++ keine Einschränkungen vor, was mit einer `register` ist. Es ist beispielsweise gültig, die Adresse einer `register` zu übernehmen, dies kann jedoch verhindern, dass der Compiler eine solche Variable tatsächlich in einem Register speichert.

C ++ 17

Das Schlüsselwort - `register` wird nicht verwendet und reserviert. Ein Programm, das das Schlüsselwortregister verwendet `register` ist schlecht geformt.

statisch

Der `static` Speicherklassenspezifizierer hat drei verschiedene Bedeutungen.

1. Gibt eine interne Verknüpfung zu einer im Namespace-Bereich deklarierten Variablen oder Funktion.

   ```
   // internal function; can't be linked to
   static double semiperimeter(double a, double b, double c) {
   ```

```
    return (a + b + c)/2.0;
}
// exported to client
double area(double a, double b, double c) {
    const double s = semiperimeter(a, b, c);
    return sqrt(s*(s-a)*(s-b)*(s-c));
}
```

2. Deklariert eine Variable mit statischer Speicherdauer (sofern es sich nicht um `thread_local`). Namespace-Gültigkeitsbereichsvariablen sind implizit statisch. Eine statische lokale Variable wird nur einmal initialisiert. Das erste Mal, wenn die Steuerung ihre Definition durchläuft, wird nicht bei jedem Verlassen des Gültigkeitsbereichs zerstört.

```
void f() {
    static int count = 0;
    std::cout << "f has been called " << ++count << " times so far\n";
}
```

3. Wenn es auf die Deklaration eines Klassenmitglieds angewendet wird, wird dieses Mitglied als statisches Mitglied deklariert.

```
struct S {
    static S* create() {
        return new S;
    }
};
int main() {
    S* s = S::create();
}
```

Beachten Sie, dass im Falle eines statischen Datenelements einer Klasse sowohl 2 als auch 3 gleichzeitig gelten: Das `static` Schlüsselwort macht das Member zu einem statischen Datenmitglied und zu einer Variablen mit statischer Speicherdauer.

Auto

C ++ 03

Deklariert eine Variable für die automatische Speicherdauer. Sie ist redundant, da die automatische Speicherdauer im Blockbereich bereits Standard ist und der automatische Bezeichner im Namespace-Bereich nicht zulässig ist.

```
void f() {
    auto int x; // equivalent to: int x;
    auto y;     // illegal in C++; legal in C89
}
auto int z;     // illegal: namespace-scope variable cannot be automatic
```

In C ++ 11 hat `auto` die Bedeutung komplett geändert und ist nicht länger ein Speicherklassenspezifizierer, sondern wird stattdessen für die Typabzugung verwendet .

extern

Der `extern` Speicherklassenspezifizierer kann eine Deklaration je nach Kontext auf eine der drei folgenden Arten ändern:

1. Es kann verwendet werden, um eine Variable zu deklarieren, ohne sie zu definieren. Normalerweise wird dies in einer Headerdatei für eine Variable verwendet, die in einer separaten Implementierungsdatei definiert wird.

```
// global scope
int x;             // definition; x will be default-initialized
extern int y;      // declaration; y is defined elsewhere, most likely another TU
extern int z = 42; // definition; "extern" has no effect here (compiler may warn)
```

2. Es gibt eine externe Verknüpfung zu einer Variablen im Gültigkeitsbereich des Namespaces, auch wenn `const` oder `constexpr` die interne Verknüpfung sonst verursacht hätte.

```
// global scope
const int w = 42;            // internal linkage in C++; external linkage in C
static const int x = 42;     // internal linkage in both C++ and C
extern const int y = 42;     // external linkage in both C++ and C
namespace {
    extern const int z = 42; // however, this has internal linkage since
                             // it's in an unnamed namespace
}
```

3. Es deklariert eine Variable im Blockbereich neu, wenn sie zuvor mit Verknüpfung deklariert wurde. Andernfalls wird eine neue Variable mit Verknüpfung deklariert, die dem nächstgelegenen umschließenden Namespace angehört.

```
// global scope
namespace {
    int x = 1;
    struct C {
        int x = 2;
        void f() {
            extern int x;           // redeclares namespace-scope x
            std::cout << x << '\n'; // therefore, this prints 1, not 2
        }
    };
}
void g() {
    extern int y; // y has external linkage; refers to global y defined elsewhere
}
```

Eine Funktion kann auch `extern` deklariert werden, dies hat jedoch keine Auswirkungen. Sie wird normalerweise als Hinweis für den Leser verwendet, dass eine hier deklarierte Funktion in einer anderen Übersetzungseinheit definiert ist. Zum Beispiel:

```
void f();        // typically a forward declaration; f defined later in this TU
extern void g(); // typically not a forward declaration; g defined in another TU
```

Wenn im obigen Code `f` auf `extern` und `g` auf nicht `extern` geändert wurde, würde dies die Korrektheit oder Semantik des Programms nicht beeinträchtigen, würde aber den Leser des Codes wahrscheinlich verwirren.

Speicherklassenspezifizierer online lesen:
https://riptutorial.com/de/cplusplus/topic/9225/speicherklassenspezifizierer

Kapitel 107: Speicherverwaltung

Syntax

- : :(*opt*) new (*Ausdrucksliste*) (*opt*) *new-type-id new-initializer* (*opt*)
- : :(*opt*) new (*Ausdrucksliste*) (*opt*) (*Typ-ID*) *new-initializer* (*opt*)
- : :((*opt*) löscht den *Cast-Ausdruck*
- : :((*opt*) delete [] *Cast-Ausdruck*
- std :: unique_ptr < *Typ-ID* > Variablenname (neue *Typ-ID* (*Opt*)); // C ++ 11
- std :: shared_ptr < *Typ-ID* > Variablenname (neue *Typ-ID* (*Opt*)); // C ++ 11
- std :: shared_ptr < *typ-id* > var_name = std :: make_shared < *typ-id* > (*opt*); // C ++ 11
- std :: unique_ptr < *typ-id* > var_name = std :: make_unique < *typ-id* > (*opt*); // C ++ 14

Bemerkungen

Ein Lead `::` zwingt den New- oder Delete-Operator dazu, im globalen Gültigkeitsbereich nachzuschlagen, wobei alle überlasteten klassenspezifischen New- oder Delete-Operatoren überschrieben werden.

Die optionalen Argumente nach dem `new` Schlüsselwort werden normalerweise verwendet, um Platzierung neu aufzurufen, können jedoch auch verwendet werden, um zusätzliche Informationen an den Zuweiser zu übergeben, z. B. ein Tag, das anfordert, dass Speicher aus einem ausgewählten Pool zugewiesen wird.

Der zugewiesene Typ wird normalerweise explizit angegeben, *z. B.* `new Foo` , kann aber auch als `auto` (seit C ++ 11) oder `decltype(auto)` (seit C ++ 14) geschrieben werden, um ihn vom Initialisierer abzuleiten.

Die Initialisierung des zugeordneten Objekts erfolgt nach den gleichen Regeln wie die Initialisierung lokaler Variablen. Insbesondere wird das Objekt standardmäßig initialisiert, wenn der Initialisierer nicht angegeben wird, und wenn ein Skalar-Typ oder ein Array mit einem Skalar-Typ dynamisch zugewiesen wird, kann nicht garantiert werden, dass der Speicher auf Null gesetzt wird.

Ein mit einem *neuen Ausdruck* erstelltes Array-Objekt muss mit `delete[]` , unabhängig davon, ob der *neue Ausdruck* mit `[]` oder nicht. Zum Beispiel:

```
using IA = int[4];
int* pIA = new IA;
delete[] pIA;  // right
// delete pIA;  // wrong
```

Examples

Stapel

Der Stack ist ein kleiner Speicherbereich, in den temporäre Werte während der Ausführung eingefügt werden. Die Zuordnung von Daten in den Stapel ist im Vergleich zur Heap-Zuordnung sehr schnell, da der gesamte Speicher bereits für diesen Zweck zugewiesen wurde.

```
int main() {
    int a = 0; //Stored on the stack
    return a;
}
```

Der Stack wird benannt, weil Ketten von Funktionsaufrufen ihren temporären Speicher übereinander "stapeln", wobei jeder einen separaten kleinen Speicherbereich verwendet.

```
float bar() {
    //f will be placed on the stack after anything else
    float f = 2;
    return f;
}

double foo() {
    //d will be placed just after anything within main()
    double d = bar();
    return d;
}

int main() {
    //The stack has no user variables stored in it until foo() is called
    return (int)foo();
}
```

Auf dem Stack gespeicherte Daten sind nur gültig, solange der Bereich, der die Variable zugewiesen hat, noch aktiv ist.

```
int* pA = nullptr;

void foo() {
    int b = *pA;
    pA = &b;
}

int main() {
    int a = 5;
    pA = &a;
    foo();
    //Undefined behavior, the value pointed to by pA is no longer in scope
    a = *pA;
}
```

Freier Speicher (Heap, dynamische Zuordnung ...)

Der Begriff **"Heap"** ist ein allgemeiner Berechnungsausdruck, der einen Speicherbereich bedeutet, aus dem Teile zugewiesen und freigegeben werden können, unabhängig von dem vom **Stapel** bereitgestellten Speicher.

In C++ der *Standard* diesen Bereich als **Free Store,** was als genauerer Begriff angesehen wird.

Vom **Free Store** zugewiesene **Speicherbereiche** leben möglicherweise länger als der ursprüngliche Umfang, in dem sie zugewiesen wurden. Daten, die zu groß sind, um auf dem Stack gespeichert zu werden, können auch aus dem **Free Store** zugewiesen werden.

Rohspeicher kann durch die *neuen* und *Löschschlüsselwörter* zugewiesen und freigegeben werden.

```
float *foo = nullptr;
{
    *foo = new float; // Allocates memory for a float
    float bar;              // Stack allocated
} // End lifetime of bar, while foo still alive

delete foo;          // Deletes the memory for the float at pF, invalidating the pointer
foo = nullptr;       // Setting the pointer to nullptr after delete is often considered good
practice
```

Es ist auch möglich, Arrays fester Größe mit *new* und *delete* mit einer etwas anderen Syntax zuzuordnen. Die Arrayzuweisung ist nicht kompatibel mit der Nicht-Arrayzuordnung. Das Mischen der beiden führt zu Heap-Beschädigungen. Durch die Zuweisung eines Arrays wird auch Speicher zugewiesen, um die Größe des Arrays für eine spätere Löschung in einer implementierungsdefinierten Weise zu verfolgen.

```
// Allocates memory for an array of 256 ints
int *foo = new int[256];
// Deletes an array of 256 ints at foo
delete[] foo;
```

Wenn Sie *new* und *delete* statt *malloc* und *free verwenden* , werden Konstruktor und Destruktor ausgeführt (ähnlich wie bei stapelbasierten Objekten). Deshalb werden *Neu* und *Löschen* gegenüber *Malloc* und *kostenlos* bevorzugt.

```
struct ComplexType {
    int a = 0;

    ComplexType() { std::cout << "Ctor" << std::endl; }
    ~ComplexType() { std::cout << "Dtor" << std::endl; }
};

// Allocates memory for a ComplexType, and calls its constructor
ComplexType *foo = new ComplexType();
//Calls the destructor for ComplexType() and deletes memory for a Complextype at pC
delete foo;
```

C ++ 11

Ab C ++ 11 wird die Verwendung von intelligenten Zeigern für die Angabe des Eigentums empfohlen.

C ++ 14

C ++ 14 fügte `std::make_unique` zur STL hinzu und änderte die Empfehlung, um `std::make_unique` oder `std::make_shared` zu `std::make_shared` anstatt nacktes *new* und *delete zu verwenden* .

Platzierung neu

Es gibt Situationen, in denen wir uns beim Zuweisen von Speicher nicht auf Free Store verlassen möchten und benutzerdefinierte Speicherzuordnungen mit `new` möchten.

Für diese Situationen können wir `Placement New` , wo wir dem 'new'-Operator mitteilen können, dass er Speicherplatz von einem zuvor zugewiesenen Speicherplatz zuweisen soll

Zum Beispiel

```
int a4byteInteger;

char *a4byteChar = new (&a4byteInteger) char[4];
```

In diesem Beispiel ist der Speicher, auf den `a4byteChar` zeigt, 4 Byte, die über die Integer-Variable `a4byteInteger` dem 'Stack' `a4byteInteger` .

Der Vorteil dieser Art der Speicherzuordnung ist die Tatsache, dass Programmierer die Zuweisung steuern. Da im `a4byteInteger` Beispiel `a4byteInteger` auf Stack zugewiesen wird, müssen Sie nicht explizit "delete a4byteChar" aufrufen.

Dasselbe Verhalten kann auch für dynamisch zugewiesenen Speicher erreicht werden. Zum Beispiel

```
int *a8byteDynamicInteger = new int[2];

char *a8byteChar = new (a8byteDynamicInteger) char[8];
```

In diesem Fall verweist der Speicherzeiger von `a8byteChar` auf den dynamischen Speicher, der von `a8byteDynamicInteger` zugewiesen wird. In diesem Fall müssen Sie jedoch `delete a8byteDynamicInteger` explizit aufrufen, um den Speicher freizugeben

Ein weiteres Beispiel für die C ++ - Klasse

```
struct ComplexType {
    int a;

    ComplexType() : a(0) {}
    ~ComplexType() {}
};

int main() {
    char* dynArray = new char[256];

    //Calls ComplexType's constructor to initialize memory as a ComplexType
    new((void*)dynArray) ComplexType();

    //Clean up memory once we're done
    reinterpret_cast<ComplexType*>(dynArray)->~ComplexType();
    delete[] dynArray;

    //Stack memory can also be used with placement new
    alignas(ComplexType) char localArray[256]; //alignas() available since C++11
```

```
    new((void*)localArray) ComplexType();

    //Only need to call the destructor for stack memory
    reinterpret_cast<ComplexType*>(localArray)->~ComplexType();

    return 0;
}
```

Speicherverwaltung online lesen:
https://riptutorial.com/de/cplusplus/topic/2873/speicherverwaltung

Kapitel 108: Spezielle Mitgliederfunktionen

Examples

Virtuelle und geschützte Destruktoren

Eine Klasse, die entworfen wurde, um vererbt zu werden, wird als Basisklasse bezeichnet. Bei den speziellen Memberfunktionen einer solchen Klasse ist Vorsicht geboten.

Eine Klasse, die zur Laufzeit (durch einen Zeiger auf die Basisklasse) polymorph verwendet wird, sollte den Destruktor als `virtual` deklarieren. Dadurch können die abgeleiteten Teile des Objekts ordnungsgemäß zerstört werden, selbst wenn das Objekt durch einen Zeiger auf die Basisklasse zerstört wird.

```
class Base {
public:
    virtual ~Base() = default;

private:
    //    data members etc.
};

class Derived : public Base { //  models Is-A relationship
public:
    //    some methods

private:
    //    more data members
};

//    virtual destructor in Base ensures that derived destructors
//    are also called when the object is destroyed
std::unique_ptr<Base> base = std::make_unique<Derived>();
base = nullptr;  //    safe, doesn't leak Derived's members
```

Wenn die Klasse nicht polymorph sein muss, die Schnittstelle jedoch vererbt werden muss, verwenden Sie einen nicht virtuellen `protected` Destruktor.

```
class NonPolymorphicBase {
public:
    //    some methods

protected:
    ~NonPolymorphicBase() = default; //    note: non-virtual

private:
    //    etc.
};
```

Eine solche Klasse kann niemals durch einen Zeiger zerstört werden, um stille Undichtigkeiten durch Schneiden zu vermeiden.

Diese Technik gilt insbesondere für Klassen, die als `private` Basisklassen konzipiert wurden. Mit einer solchen Klasse können einige allgemeine Implementierungsdetails gekapselt werden, während `virtual` Methoden als Anpassungspunkte bereitgestellt werden. Diese Art von Klassen sollte niemals polymorph verwendet werden, und ein `protected` Destruktor hilft, diese Anforderung direkt im Code zu dokumentieren.

Schließlich können einige Klassen verlangen, dass sie *niemals* als Basisklasse verwendet werden. In diesem Fall kann die Klasse als `final` markiert werden. In diesem Fall ist ein normaler nicht virtueller öffentlicher Destruktor in Ordnung.

```
class FinalClass final {  //    marked final here
public:
    ~FinalClass() = default;

private:
    //    etc.
};
```

Implizites Verschieben und Kopieren

Beachten Sie, dass das Deklarieren eines Destruktors den Compiler daran hindert, implizite Verschiebungskonstruktoren und Verschiebungszuweisungsoperatoren zu generieren. Wenn Sie einen Destruktor deklarieren, müssen Sie auch die entsprechenden Definitionen für die Verschiebungsoperationen hinzufügen.

Durch das Deklarieren von Verschiebungsvorgängen wird außerdem die Erzeugung von Kopiervorgängen unterdrückt. Diese sollten ebenfalls hinzugefügt werden (wenn die Objekte dieser Klasse Kopiersemantik benötigen).

```
class Movable {
public:
    virtual ~Movable() noexcept = default;

    //    compiler won't generate these unless we tell it to
    //    because we declared a destructor
    Movable(Movable&&) noexcept = default;
    Movable& operator=(Movable&&) noexcept = default;

    //    declaring move operations will suppress generation
    //    of copy operations unless we explicitly re-enable them
    Movable(const Movable&) = default;
    Movable& operator=(const Movable&) = default;
};
```

Kopieren und tauschen

Wenn Sie eine Klasse schreiben, die Ressourcen verwaltet, müssen Sie alle speziellen Member-Funktionen implementieren (siehe Drei- / Fünf-Regel / Null). Der direkteste Ansatz zum Schreiben des Kopierkonstruktors und des Zuweisungsoperators wäre:

```
person(const person &other)
```

```
    : name(new char[std::strlen(other.name) + 1])
    , age(other.age)
{
    std::strcpy(name, other.name);
}

person& operator=(person const& rhs) {
    if (this != &other) {
        delete [] name;
        name = new char[std::strlen(other.name) + 1];
        std::strcpy(name, other.name);
        age = other.age;
    }

    return *this;
}
```

Dieser Ansatz hat jedoch einige Probleme. Die starke Ausnahmegarantie fällt nicht aus - wenn `new[]` wirft, haben wir die Ressourcen, die sich im Besitz `this` befinden, bereits gelöscht und können sie nicht wiederherstellen. Wir duplizieren viel von der Logik der Kopierkonstruktion bei der Kopierzuweisung. Und wir müssen uns an die Selbstzuweisungsprüfung erinnern, die den Kopiervorgang normalerweise nur belastet, aber immer noch kritisch ist.

Um die starke Ausnahmegarantie zu erfüllen und die Codevervielfältigung zu vermeiden (doppelt so mit dem nachfolgenden Verschiebungszuweisungsoperator), können wir das Copy-and-Swap-Idiom verwenden:

```
class person {
    char* name;
    int age;
public:
    /* all the other functions ... */

    friend void swap(person& lhs, person& rhs) {
        using std::swap; // enable ADL

        swap(lhs.name, rhs.name);
        swap(lhs.age, rhs.age);
    }

    person& operator=(person rhs) {
        swap(*this, rhs);
        return *this;
    }
};
```

Warum funktioniert das? Überlegen Sie, was passiert, wenn wir haben

```
person p1 = ...;
person p2 = ...;
p1 = p2;
```

Zuerst kopieren wir `rhs` aus `p2` (was wir hier nicht duplizieren mussten). Wenn diese Operation wirft, machen wir nichts in `operator=` und `p1` bleibt unangetastet. Als nächstes tauschen wir die Mitglieder zwischen `*this` und `rhs` , und dann geht `rhs` außer Reichweite. Wenn `operator=` , reinigt ,

dass implizit die ursprünglichen Ressourcen `this` (über den destructor, die wir nicht duplizieren müssen). Selbstzuweisung funktioniert auch - mit Copy-and-Swap ist dies weniger effizient (erfordert eine zusätzliche Zuweisung und Freigabe), aber in diesem unwahrscheinlichen Fall verlangsamen wir den typischen Anwendungsfall nicht, um dies zu berücksichtigen.

C ++ 11

Die obige Formulierung funktioniert bereits für die Bewegungszuweisung.

```
p1 = std::move(p2);
```

Hier verschieben wir `rhs` von `p2` , und der Rest ist genauso gültig. Wenn eine Klasse verschiebbar, aber nicht kopierbar ist, muss die Kopierzuweisung nicht gelöscht werden, da dieser Zuweisungsoperator aufgrund des gelöschten Kopierkonstruktors einfach falsch formatiert wird.

Standardkonstruktor

Ein *Standardkonstruktor* ist ein Typ eines Konstruktors, der beim Aufruf keine Parameter erfordert. Es ist nach dem von ihm konstruierten Typ benannt und ist eine Memberfunktion (wie alle Konstruktoren).

```
class C{
    int i;
public:
    // the default constructor definition
    C()
    : i(0){ // member initializer list -- initialize i to 0
        // constructor function body -- can do more complex things here
    }
};

C c1; // calls default constructor of C to create object c1
C c2 = C(); // calls default constructor explicitly
C c3(); // ERROR: this intuitive version is not possible due to "most vexing parse"
C c4{}; // but in C++11 {} CAN be used in a similar way

C c5[2]; // calls default constructor for both array elements
C* c6 = new C[2]; // calls default constructor for both array elements
```

Eine andere Möglichkeit, die Anforderung "keine Parameter" zu erfüllen, besteht darin, dass der Entwickler Standardwerte für alle Parameter bereitstellt:

```
class D{
    int i;
    int j;
public:
    // also a default constructor (can be called with no parameters)
    D( int i = 0, int j = 42 )
    : i(i), j(j){
    }
};
```

```
D d; // calls constructor of D with the provided default values for the parameters
```

Unter bestimmten Umständen (dh der Entwickler stellt keine Konstruktoren bereit und es gibt keine anderen disqualifizierenden Bedingungen), stellt der Compiler implizit einen leeren Standardkonstruktor bereit:

```
class C{
    std::string s; // note: members need to be default constructible themselves
};

C c1; // will succeed -- C has an implicitly defined default constructor
```

Einen anderen Konstruktor-Typ zu haben, ist eine der oben genannten Disqualifizierungsbedingungen:

```
class C{
    int i;
public:
    C( int i ) : i(i){}
};

C c1; // Compile ERROR: C has no (implicitly defined) default constructor
```

c ++ 11

Um die Erstellung eines impliziten Standardkonstruktors zu verhindern, ist es üblich, ihn als `private` zu deklarieren (ohne Definition). Die Absicht ist, einen Kompilierungsfehler zu verursachen, wenn jemand versucht, den Konstruktor zu verwenden (dies führt je nach Compiler entweder zu einem *Zugriff auf private* Fehler oder zu einem Linker-Fehler).

Um sicher zu sein, dass ein Standardkonstruktor (der dem impliziten funktionell ähnlich ist) definiert ist, könnte ein Entwickler explizit einen leeren Konstruktor schreiben.

c ++ 11

In C ++ 11 kann ein Entwickler auch das Schlüsselwort `delete` , um zu verhindern, dass der Compiler einen Standardkonstruktor bereitstellt.

```
class C{
    int i;
public:
    // default constructor is explicitly deleted
    C() = delete;
};

C c1; // Compile ERROR: C has its default constructor deleted
```

Darüber hinaus kann ein Entwickler auch explizit angeben, dass der Compiler einen Standardkonstruktor bereitstellen soll.

```
class C{
    int i;
```

```
public:
    // does have automatically generated default constructor (same as implicit one)
    C() = default;

    C( int i ) : i(i){}
};

C c1; // default constructed
C c2( 1 ); // constructed with the int taking constructor
```

c ++ 14

Sie können mithilfe von `std::is_default_constructible` aus `<type_traits>` bestimmen, ob ein Typ über einen Standardkonstruktor verfügt (oder ein primitiver Typ ist):

```
class C1{ };
class C2{ public: C2(){} };
class C3{ public: C3(int){} };

using std::cout; using std::boolalpha; using std::endl;
using std::is_default_constructible;
cout << boolalpha << is_default_constructible<int>() << endl; // prints true
cout << boolalpha << is_default_constructible<C1>() << endl; // prints true
cout << boolalpha << is_default_constructible<C2>() << endl; // prints true
cout << boolalpha << is_default_constructible<C3>() << endl; // prints false
```

c ++ 11

In C ++ 11 ist es weiterhin möglich, die Nicht-Funktionsversion von `std::is_default_constructible` :

```
cout << boolalpha << is_default_constructible<C1>::value << endl; // prints true
```

Zerstörer

Ein *Destruktor* ist eine Funktion ohne Argumente, die aufgerufen wird, wenn ein benutzerdefiniertes Objekt zerstört werden soll. Es ist nach dem Typ benannt, den es mit einem `~` Präfix zerstört.

```
class C{
    int* is;
    string s;
public:
    C()
    : is( new int[10] ){
    }

    ~C(){  // destructor definition
        delete[] is;
    }
};

class C_child : public C{
    string s_ch;
public:
    C_child(){}
```

```
    ~C_child(){} // child destructor
};

void f(){
    C c1; // calls default constructor
    C c2[2]; // calls default constructor for both elements
    C* c3 = new C[2]; // calls default constructor for both array elements

    C_child c_ch;  // when destructed calls destructor of s_ch and of C base (and in turn s)

    delete[] c3; // calls destructors on c3[0] and c3[1]
} // automatic variables are destroyed here -- i.e. c1, c2 and c_ch
```

In den meisten Fällen (dh ein Benutzer stellt keinen Destruktor bereit, und es gibt keine anderen disqualifizierenden Bedingungen), stellt der Compiler implizit einen Standard-Destruktor bereit:

```
class C{
    int i;
    string s;
};

void f(){
    C* c1 = new C;
    delete c1; // C has a destructor
}
```

```
class C{
    int m;
private:
    ~C(){} // not public destructor!
};

class C_container{
    C c;
};

void f(){
    C_container* c_cont = new C_container;
    delete c_cont; // Compile ERROR: C has no accessible destructor
}
```

c ++ 11

In C ++ 11 kann ein Entwickler dieses Verhalten überschreiben, indem er verhindert, dass der Compiler einen Standard-Destruktor bereitstellt.

```
class C{
    int m;
public:
    ~C() = delete; // does NOT have implicit destructor
};

void f{
    C c1;
} // Compile ERROR: C has no destructor
```

Darüber hinaus kann ein Entwickler auch explizit angeben, dass der Compiler einen Standard-Destruktor bereitstellen soll.

```
class C{
    int m;
public:
    ~C() = default; // saying explicitly it does have implicit/empty destructor
};

void f(){
    C c1;
} // C has a destructor -- c1 properly destroyed
```

c ++ 11

Sie können mit `std::is_destructible` aus `<type_traits>` feststellen, ob ein Typ einen Destruktor hat (oder ein primitiver Typ ist):

```
class C1{ };
class C2{ public: ~C2() = delete };
class C3 : public C2{ };

using std::cout; using std::boolalpha; using std::endl;
using std::is_destructible;
cout << boolalpha << is_destructible<int>() << endl; // prints true
cout << boolalpha << is_destructible<C1>() << endl; // prints true
cout << boolalpha << is_destructible<C2>() << endl; // prints false
cout << boolalpha << is_destructible<C3>() << endl; // prints false
```

Spezielle Mitgliederfunktionen online lesen:
https://riptutorial.com/de/cplusplus/topic/1476/spezielle-mitgliederfunktionen

Kapitel 109: Standard-Bibliotheksalgorithmen

Examples

std :: for_each

```
template<class InputIterator, class Function>
    Function for_each(InputIterator first, InputIterator last, Function f);
```

Auswirkungen:

Wendet `f` auf das Ergebnis der Dereferenzierung jedes Iterators im Bereich `[first, last)` beginnend mit dem `first` und dem `last - 1` .

Parameter:

`first, last` - der Bereich, auf den `f` soll.

`f` - Callable Object, das auf das Ergebnis der Dereferenzierung jedes Iterators im Bereich `[first, last)` angewendet wird.

Rückgabewert:

`f` (bis C ++ 11) und `std::move(f)` (seit C ++ 11).

Komplexität:

Wendet `f` genau das `last - first` Mal an.

Beispiel:

c ++ 11

```
std::vector<int> v { 1, 2, 4, 8, 16 };
std::for_each(v.begin(), v.end(), [](int elem) { std::cout << elem << " "; });
```

Wendet die angegebene Funktion für jedes Element des Vektors `v` , das dieses Element auf `stdout` .

std :: next_permutation

```
template< class Iterator >
bool next_permutation( Iterator first, Iterator last );
template< class Iterator, class Compare >
bool next_permutation( Iterator first, Iterator last, Compare cmpFun );
```

Auswirkungen:
Verschieben Sie die Datenfolge des Bereichs [first, last] in die nächste lexikographisch höhere

Permutation. Wenn `cmpFun` bereitgestellt wird, wird die Permutationsregel angepasst.

Parameter:
`first` - der Beginn des zu permutierenden Bereichs einschließlich
`last` - das Ende des zu permutierenden Bereichs, exklusiv

Rückgabewert:
Gibt true zurück, wenn eine solche Permutation vorhanden ist.
Andernfalls wird der Bereich auf die lexikographisch kleinste Permutation getauscht und false zurückgegeben.

Komplexität:
O (n), n ist der Abstand vom `first` bis zum `last` .

Beispiel :

```
std::vector< int > v { 1, 2, 3 };
do
{
   for( int i = 0; i < v.size(); i += 1 )
   {
       std::cout << v[i];
   }
   std::cout << std::endl;
}while( std::next_permutation( v.begin(), v.end() ) );
```

Drucke alle Permutationsfälle von 1,2,3 in lexikographisch ansteigender Reihenfolge.
Ausgabe:

```
123
132
213
231
312
321
```

std :: akkumulieren

Definiert in Kopfzeile `<numeric>`

```
template<class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init); // (1)

template<class InputIterator, class T, class BinaryOperation>
T accumulate(InputIterator first, InputIterator last, T init, BinaryOperation f); // (2)
```

Auswirkungen:

std :: Accumulate führt eine Fold- Operation aus, wobei die `f` Funktion im Bereich `[first, last)` beginnend mit `init` als Akkumulatorwert verwendet wird.

Im Grunde entspricht es:

```
T acc = init;
for (auto it = first; first != last; ++it)
    acc = f(acc, *it);
return acc;
```

In Version (1) wird der `operator+` anstelle von `f` , sodass das Sammeln über Container der Summe der Containerelemente entspricht.

Parameter:

`first, last` - der Bereich, auf den `f` soll.
`init` - Anfangswert des Akkus.
`f` - binäre Faltfunktion.

Rückgabewert:

Gesamtwert der `f` Anwendungen.

Komplexität:

O (n × k) , wobei *n* der Abstand vom `first` bis zum `last` , *O (k)* die Komplexität der `f` Funktion.

Beispiel:

Einfaches Summenbeispiel:

```
std::vector<int> v { 2, 3, 4 };
auto sum = std::accumulate(v.begin(), v.end(), 1);
std::cout << sum << std::endl;
```

Ausgabe:

```
10
```

Ziffern in Zahl umrechnen:

c ++ 11

```
class Converter {
public:
    int operator()(int a, int d) const { return a * 10 + d; }
};
```

und später

```
const int ds[3] = {1, 2, 3};
int n = std::accumulate(ds, ds + 3, 0, Converter());
std::cout << n << std::endl;
```

c ++ 11

```
const std::vector<int> ds = {1, 2, 3};
int n = std::accumulate(ds.begin(), ds.end(),
                        0,
                        [](int a, int d) { return a * 10 + d; });
std::cout << n << std::endl;
```

Ausgabe:

```
123
```

std :: find

```
template <class InputIterator, class T>
InputIterator find (InputIterator first, InputIterator last, const T& val);
```

Auswirkungen

Findet das erste Vorkommen von val innerhalb des Bereichs [first, last)

Parameter

`first` => Iterator zeigt auf den Anfang des Bereichs. `last` => Iterator zeigt auf das Ende des Bereichs. `val` => Der zu `val` Wert innerhalb des Bereichs

Rückkehr

Ein Iterator, der auf das erste Element innerhalb des Bereichs zeigt, der gleich (==) mit val ist. Der Iterator zeigt auf das letzte Element, wenn val nicht gefunden wird.

Beispiel

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;

int main(int argc, const char * argv[]) {

  //create a vector
  vector<int> intVec {4, 6, 8, 9, 10, 30, 55,100, 45, 2, 4, 7, 9, 43, 48};

  //define iterators
  vector<int>::iterator  itr_9;
  vector<int>::iterator  itr_43;
  vector<int>::iterator  itr_50;

  //calling find
  itr_9 = find(intVec.begin(), intVec.end(), 9); //occurs twice
  itr_43 = find(intVec.begin(), intVec.end(), 43); //occurs once

  //a value not in the vector
  itr_50 = find(intVec.begin(), intVec.end(), 50); //does not occur
```

```
  cout << "first occurence of: " << *itr_9 << endl;
  cout << "only occurence of: " << *itr_43 << Lendl;


  /*
    let's prove that itr_9 is pointing to the first occurence
    of 9 by looking at the element after 9, which should be 10
    not 43
  */
  cout << "element after first 9: " << *(itr_9 + 1) << ends;

  /*
    to avoid dereferencing intVec.end(), lets look at the
    element right before the end
  */
  cout << "last element: " << *(itr_50 - 1) << endl;

  return 0;
}
```

Ausgabe

```
first occurence of: 9
only occurence of: 43
element after first 9: 10
last element: 48
```

std :: count

```
template <class InputIterator, class T>
typename iterator_traits<InputIterator>::difference_type
count (InputIterator first, InputIterator last, const T& val);
```

Auswirkungen

Zählt die Anzahl der Elemente, die val entsprechen

Parameter

`first` => Iterator zeigt auf den Anfang des Bereichs
`last` => Iterator zeigt auf das Ende des Bereichs
`val` => Das Vorkommen dieses Wertes im Bereich wird gezählt

Rückkehr

Die Anzahl der Elemente im Bereich, die gleich (==) bis val sind.

Beispiel

```
#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;
```

```
int main(int argc, const char * argv[]) {

  //create vector
  vector<int> intVec{4,6,8,9,10,30,55,100,45,2,4,7,9,43,48};

  //count occurences of 9, 55, and 101
  size_t count_9 = count(intVec.begin(), intVec.end(), 9); //occurs twice
  size_t count_55 = count(intVec.begin(), intVec.end(), 55); //occurs once
  size_t count_101 = count(intVec.begin(), intVec.end(), 101); //occurs once

  //print result
  cout << "There are " << count_9  << " 9s"<< endl;
  cout << "There is " << count_55  << " 55"<< endl;
  cout << "There is " << count_101  << " 101"<< ends;

  //find the first element == 4 in the vector
  vector<int>::iterator itr_4 = find(intVec.begin(), intVec.end(), 4);

  //count its occurences in the vector starting from the first one
  size_t count_4 = count(itr_4, intVec.end(), *itr_4); // should be 2

  cout << "There are " << count_4  << " " << *itr_4 << endl;

  return 0;
}
```

Ausgabe

```
There are 2 9s
There is 1 55
There is 0 101
There are 2 4
```

std :: count_if

```
template <class InputIterator, class UnaryPredicate>
typename iterator_traits<InputIterator>::difference_type
count_if (InputIterator first, InputIterator last, UnaryPredicate red);
```

Auswirkungen

Zählt die Anzahl der Elemente in einem Bereich, für die eine angegebene Prädikatfunktion wahr ist

Parameter

`first` => Iterator zum Anfang des Bereichs zeigt `last` => Iterator auf das Ende des Bereichs zeigt `red` => Prädikatfunktion (gibt wahr oder falsch)

Rückkehr

Die Anzahl der Elemente innerhalb des angegebenen Bereichs, für die die Prädikatfunktion true zurückgegeben hat.

Beispiel

```
#include <iostream>
#include <vector>
#include <algorithm>

using namespace std;

/*
    Define a few functions to use as predicates
*/

//return true if number is odd
bool isOdd(int i){
  return i%2 == 1;
}

//functor that returns true if number is greater than the value of the constructor parameter
provided
class Greater {
  int _than;
public:
  Greater(int th): _than(th){}
  bool operator()(int i){
    return i > _than;
  }
};

int main(int argc, const char * argv[]) {

  //create a vector
  vector<int> myvec = {1,5,8,0,7,6,4,5,2,1,5,0,6,9,7};

  //using a lambda function to count even numbers
  size_t evenCount = count_if(myvec.begin(), myvec.end(), [](int i){return i % 2 == 0;}); //
>= C++11

  //using function pointer to count odd number in the first half of the vector
  size_t oddCount = count_if(myvec.begin(), myvec.end()- myvec.size()/2, isOdd);

  //using a functor to count numbers greater than 5
  size_t greaterCount = count_if(myvec.begin(), myvec.end(), Greater(5));

  cout << "vector size: " << myvec.size() << endl;
  cout << "even numbers: " << evenCount << " found" << endl;
  cout << "odd numbers: " << oddCount << " found" << endl;
  cout << "numbers > 5: " << greaterCount << " found"<< endl;

  return 0;
}
```

Ausgabe

```
vector size: 15
even numbers: 7 found
odd numbers: 4 found
numbers > 5: 6 found
```

std :: find_if

```
template <class InputIterator, class UnaryPredicate>
InputIterator find_if (InputIterator first, InputIterator last, UnaryPredicate pred);
```

Auswirkungen

Findet das erste Element in einem Bereich, für das die Prädikatfunktion `pred` true zurückgibt.

Parameter

`first` => Iterator zum Anfang des Bereichs zeigt `last` => Iterator auf das Ende des Bereichs zeigt `pred` => Prädikatfunktion (gibt wahr oder falsch)

Rückkehr

Ein Iterator, der auf das erste Element innerhalb des Bereichs zeigt, für den die Prädikatfunktion pred true zurückgibt. Der Iterator zeigt auf den letzten Punkt, wenn val nicht gefunden wird

Beispiel

```
#include <iostream>
#include <vector>
#include <algorithm>


using namespace std;

/*
    define some functions to use as predicates
*/

//Returns true if x is multiple of 10
bool multOf10(int x) {
  return x % 10 == 0;
}

//returns true if item greater than passed in parameter
class Greater {
  int _than;

public:
  Greater(int th):_than(th){

  }
  bool operator()(int data) const
  {
    return data > _than;
  }
};


int main()
{

  vector<int> myvec {2, 5, 6, 10, 56, 7, 48, 89, 850, 7, 456};
```

```
  //with a lambda function
  vector<int>::iterator gt10 = find_if(myvec.begin(), myvec.end(), [](int x){return x>10;});
// >= C++11

  //with a function pointer
  vector<int>::iterator pow10 = find_if(myvec.begin(), myvec.end(), multOf10);

  //with functor
  vector<int>::iterator gt5 = find_if(myvec.begin(), myvec.end(), Greater(5));

  //not Found
  vector<int>::iterator nf = find_if(myvec.begin(), myvec.end(), Greater(1000)); // nf points
to myvec.end()


  //check if pointer points to myvec.end()
  if(nf != myvec.end()) {
    cout << "nf points to: " << *nf << endl;
  }
  else {
    cout << "item not found" << endl;
  }



  cout << "First item >   10: " << *gt10  << endl;
  cout << "First Item n * 10: " << *pow10 << endl;
  cout << "First Item >    5: " << *gt5   << endl;

  return 0;
}
```

Ausgabe

```
item not found
First item >   10: 56
First Item n * 10: 10
First Item >    5: 6
```

std :: min_element

```
template <class ForwardIterator>
ForwardIterator min_element (ForwardIterator first, ForwardIterator last);

template <class ForwardIterator, class Compare>
ForwardIterator min_element (ForwardIterator first, ForwardIterator last,Compare comp);
```

Auswirkungen

Findet das minimale Element in einem Bereich

Parameter

`first` Iterator zeigt auf den Anfang des Bereichs
`last` iterator, der auf das Ende des Bereichs `comp` - ein Funktionszeiger oder ein Funktionsobjekt,

das zwei Argumente verwendet und true oder false zurückgibt, um anzugeben, ob das Argument weniger als Argument 2 ist

Rückkehr

Iterator auf das minimale Element im Bereich

Komplexität

Linear in einem weniger als die Anzahl der verglichenen Elemente.

Beispiel

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <utility>  //to use make_pair

using namespace std;

//function compare two pairs
bool pairLessThanFunction(const pair<string, int> &p1, const pair<string, int> &p2)
{
  return p1.second < p2.second;
}

int main(int argc, const char * argv[]) {

  vector<int> intVec {30,200,167,56,75,94,10,73,52,6,39,43};

  vector<pair<string, int>> pairVector = {make_pair("y", 25), make_pair("b", 2),
make_pair("z", 26), make_pair("e", 5) };


  // default using < operator
  auto minInt = min_element(intVec.begin(), intVec.end());

  //Using pairLessThanFunction
  auto minPairFunction = min_element(pairVector.begin(), pairVector.end(),
pairLessThanFunction);


  //print minimum of intVector
  cout << "min int from default: " << *minInt << endl;

  //print minimum of pairVector
  cout << "min pair from PairLessThanFunction: " << (*minPairFunction).second << endl;

  return 0;
}
```

Ausgabe

```
min int from default: 6
min pair from PairLessThanFunction: 2
```

Std :: nth_element verwenden, um den Median (oder andere Quantile) zu finden

Der `std::nth_element` Algorithmus benötigt drei Iteratoren: einen Iterator an den Anfang, die *n*- te Position und das Ende. Sobald die Funktion zurückkehrt, ist das *n*- te Element (in Reihenfolge) das *n*- te kleinste Element. (Die Funktion hat aufwendigere Überladungen, z. B. einige Vergleichsfunktionen; der Variationsbereich ist oben angegeben.)

Hinweis Diese Funktion ist sehr effizient - sie ist linear komplex.

In diesem Beispiel definieren wir den Median einer Sequenz der Länge *n* als das Element, das sich in Position ⌜n / 2⌝ befinden würde. Zum Beispiel ist der Median einer Sequenz der Länge 5 das drittkleinste Element und ebenso der Median einer Sequenz der Länge 6.

Um diese Funktion zum Finden des Medians zu verwenden, können wir Folgendes verwenden. Sagen wir, wir fangen mit an

```
std::vector<int> v{5, 1, 2, 3, 4};

std::vector<int>::iterator b = v.begin();
std::vector<int>::iterator e = v.end();

std::vector<int>::iterator med = b;
std::advance(med, v.size() / 2);

// This makes the 2nd position hold the median.
std::nth_element(b, med, e);

// The median is now at v[2].
```

Um das *p*- te Quantil zu finden , würden wir einige der obigen Zeilen ändern:

```
const std::size_t pos = p * std::distance(b, e);

std::advance(nth, pos);
```

und suchen Sie das Quantil an Position `pos` .

Standard-Bibliotheksalgorithmen online lesen:
https://riptutorial.com/de/cplusplus/topic/3177/standard-bibliotheksalgorithmen

Kapitel 110: static_assert

Syntax

- static_assert (*bool_constexpr* , *message*)
- static_assert (*bool_constexpr*) / * seit C ++ 17 * /

Parameter

Parameter	Einzelheiten
bool_constexpr	Ausdruck zum Überprüfen
Botschaft	Nachricht, die gedruckt werden soll, wenn *bool_constexpr falsch ist*

Bemerkungen

Im Gegensatz zu Laufzeit-Assertions werden statische Assertions zur Kompilierzeit geprüft und auch beim Kompilieren optimierter Builds erzwungen.

Examples

static_assert

Assertationen bedeuten, dass eine Bedingung geprüft werden muss, und wenn sie falsch ist, handelt es sich um einen Fehler. Bei `static_assert()` wird dies zur Kompilierzeit gemacht.

```
template<typename T>
T mul10(const T t)
{
    static_assert( std::is_integral<T>::value, "mul10() only works for integral types" );
    return (t << 3) + (t << 1);
}
```

Ein `static_assert()` hat einen obligatorischen ersten Parameter, die Bedingung, dh ein bool constexpr. Es *kann* einen zweiten Parameter enthalten, die Nachricht, also ein Zeichenkettenliteral. In C ++ 17 ist der zweite Parameter optional. davor ist es obligatorisch.

C ++ 17

```
template<typename T>
T mul10(const T t)
{
    static_assert(std::is_integral<T>::value);
    return (t << 3) + (t << 1);
}
```

Es wird verwendet, wenn:

- Im Allgemeinen ist eine Überprüfung zur Kompilierungszeit für einen Typ auf den Wert "constexpr" erforderlich
- Eine Vorlagenfunktion muss bestimmte Eigenschaften eines an sie übergebenen Typs überprüfen
- Man möchte Testfälle schreiben für:
    - Vorlagen-Metafunktionen
    - Constexpr-Funktionen
    - Makro-Metaprogrammierung
- Bestimmte Definitionen sind erforderlich (z. B. C ++ - Version)
- Portierung von Legacy-Code, Zusicherungen zu `sizeof(T)` (z. B. 32-Bit-Int)
- Bestimmte Compiler-Funktionen sind für das Programm erforderlich (Packen, leere Basisklassenoptimierung usw.).

Beachten Sie, dass `static_assert()` nicht an SFINAE beteiligt ist. Wenn zusätzliche Überladungen / Spezialisierungen möglich sind, sollten Sie sie nicht anstelle von Metaprogrammierungsmethoden für Vorlagen (wie `std::enable_if<>`) verwenden. Es kann im Vorlagencode verwendet werden, wenn die erwartete Überlastung / Spezialisierung bereits gefunden wurde, weitere Überprüfungen jedoch erforderlich sind. In solchen Fällen kann es konkretere Fehlermeldungen geben, als sich auf SFINAE zu verlassen.

static_assert online lesen: https://riptutorial.com/de/cplusplus/topic/3822/static-assert

Kapitel 111: std :: any

Bemerkungen

Die Klasse `std::any` stellt einen typsicheren Container bereit, in den Einzelwerte eines beliebigen Typs eingegeben werden können.

Examples

Grundlegende Verwendung

```
std::any an_object{ std::string("hello world") };
if (an_object.has_value()) {
    std::cout << std::any_cast<std::string>(an_object) << '\n';
}

try {
  std::any_cast<int>(an_object);
} catch(std::bad_any_cast&) {
    std::cout << "Wrong type\n";
}

std::any_cast<std::string&>(an_object) = "42";
std::cout << std::any_cast<std::string>(an_object) << '\n';
```

Ausgabe

```
hello world
Wrong type
42
```

std :: any online lesen: https://riptutorial.com/de/cplusplus/topic/7894/std----any

Kapitel 112: std :: array

Parameter

Parameter	Definition
`class T`	Gibt den Datentyp der Arraymitglieder an
`std::size_t N`	Gibt die Anzahl der Mitglieder im Array an

Bemerkungen

Die Verwendung eines `std::array` erfordert die Einbeziehung des `<array>` -Headers mit `#include <array>` .

Examples

Initialisieren eines std :: -Arrays

Initialisierung von `std::array<T, N>` , wobei `T` ein Skalartyp und `N` die Anzahl der Elemente vom Typ `T`

Wenn `T` ein Skalartyp ist, kann `std::array` auf folgende Weise initialisiert werden:

```
// 1) Using aggregate-initialization
std::array<int, 3> a{ 0, 1, 2 };
// or equivalently
std::array<int, 3> a = { 0, 1, 2 };

// 2) Using the copy constructor
std::array<int, 3> a{ 0, 1, 2 };
std::array<int, 3> a2(a);
// or equivalently
std::array<int, 3> a2 = a;

// 3) Using the move constructor
std::array<int, 3> a = std::array<int, 3>{ 0, 1, 2 };
```

Initialisierung von `std::array<T, N>` , wobei `T` ein nicht skalarer Typ ist und `N` die Anzahl der Elemente vom Typ `T`

Wenn `T` ein nicht-skalarer Typ ist, kann `std::array` auf folgende Weise initialisiert werden:

```
struct A { int values[3]; }; // An aggregate type

// 1) Using aggregate initialization with brace elision
// It works only if T is an aggregate type!
```

```
std::array<A, 2> a{ 0, 1, 2, 3, 4, 5 };
// or equivalently
std::array<A, 2> a = { 0, 1, 2, 3, 4, 5 };

// 2) Using aggregate initialization with brace initialization of sub-elements
std::array<A, 2> a{ A{ 0, 1, 2 }, A{ 3, 4, 5 } };
// or equivalently
std::array<A, 2> a = { A{ 0, 1, 2 }, A{ 3, 4, 5 } };

// 3)
std::array<A, 2> a{{ { 0, 1, 2 }, { 3, 4, 5 } }};
// or equivalently
std::array<A, 2> a = {{ { 0, 1, 2 }, { 3, 4, 5 } }};

// 4) Using the copy constructor
std::array<A, 2> a{ 1, 2, 3 };
std::array<A, 2> a2(a);
// or equivalently
std::array<A, 2> a2 = a;

// 5) Using the move constructor
std::array<A, 2> a = std::array<A, 2>{ 0, 1, 2, 3, 4, 5 };
```

Elementzugriff

1. `at(pos)`

Gibt einen Verweis auf das Element an Position `pos` mit Begrenzungsprüfung zurück. Wenn `pos` nicht im Bereich des Containers liegt, wird eine Ausnahme vom Typ `std::out_of_range` ausgelöst.

Die Komplexität ist konstant O (1).

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    arr.at(0) = 2;
    arr.at(1) = 4;
    arr.at(2) = 6;

    // read values
    int a = arr.at(0); // a is now 2
    int b = arr.at(1); // b is now 4
    int c = arr.at(2); // c is now 6

    return 0;
}
```

2) `operator[pos]`

Gibt einen Verweis auf das Element an der Position `pos` ohne Prüfung der Grenzen zurück. Wenn `pos` nicht im Bereich des Containers liegt, kann ein Fehler bei der *Segmentierungsverletzung der* Laufzeit auftreten. Diese Methode bietet einen Elementzugriff, der klassischen Arrays entspricht

und effizienter ist als `at(pos)` .

Die Komplexität ist konstant O (1).

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    arr[0] = 2;
    arr[1] = 4;
    arr[2] = 6;

    // read values
    int a = arr[0]; // a is now 2
    int b = arr[1]; // b is now 4
    int c = arr[2]; // c is now 6

    return 0;
}
```

3) `std::get<pos>`

Diese **Nicht-Member-** Funktion gibt einen Verweis auf das Element an **der konstanten** Position `pos` **Kompilierungszeit zurück,** ohne dass eine Prüfung der Grenzen erfolgt. Wenn `pos` nicht im Bereich des Containers liegt, kann ein Fehler bei der *Segmentierungsverletzung der* Laufzeit auftreten.

Die Komplexität ist konstant O (1).

```
#include <array>

int main()
{
    std::array<int, 3> arr;

    // write values
    std::get<0>(arr) = 2;
    std::get<1>(arr) = 4;
    std::get<2>(arr) = 6;

    // read values
    int a = std::get<0>(arr); // a is now 2
    int b = std::get<1>(arr); // b is now 4
    int c = std::get<2>(arr); // c is now 6

    return 0;
}
```

4) `front()`

Gibt einen Verweis auf das erste Element im Container zurück. Das Aufrufen von `front()` auf einem leeren Container ist undefiniert.

Die Komplexität ist konstant O (1).

Hinweis: Für einen Container c entspricht der Ausdruck `c.front() *c.begin()` .

```
#include <array>

int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };

    int a = arr.front(); // a is now 2

    return 0;
}
```

5) `back()`

Gibt den Verweis auf das letzte Element im Container zurück. Rückruf `back()` eines leeren Containers ist undefiniert.

Die Komplexität ist konstant O (1).

```
#include <array>

int main()
{
    std::array<int, 3> arr{ 2, 4, 6 };

    int a = arr.back(); // a is now 6

    return 0;
}
```

6) `data()`

Gibt einen Zeiger auf das zugrunde liegende Array zurück, das als Elementspeicher dient. Der Zeiger ist so, dass der `range [data(); data() + size())` ist immer ein gültiger Bereich, auch wenn der Container leer ist (`data()` ist in diesem Fall nicht dereferenzierbar).

Die Komplexität ist konstant O (1).

```
#include <iostream>
#include <cstring>
#include <array>

int main ()
{
    const char* cstr = "Test string";
    std::array<char, 12> arr;

    std::memcpy(arr.data(), cstr, 12); // copy cstr to arr

    std::cout << arr.data(); // outputs: Test string

    return 0;
```

```
}
```

Größe des Arrays überprüfen

Einer der Hauptvorteile von `std::array` im Vergleich zu einem `C` Stil-Array besteht darin, dass wir die Größe des Arrays mithilfe der Mitgliedsfunktion `size()` überprüfen können

```
int main() {
    std::array<int, 3> arr = { 1, 2, 3 };
    cout << arr.size() << endl;
}
```

Iteration durch das Array

`std::array` ist ein STL-Container und kann eine bereichsbasierte Schleife verwenden, die anderen Containern wie `vector` ähnelt

```
int main() {
     std::array<int, 3> arr = { 1, 2, 3 };
     for (auto i : arr)
         cout << i << '\n';
}
```

Alle Array-Elemente auf einmal ändern

Die Memberfunktion `fill()` kann für `std::array` um die Werte nach der Initialisierung sofort zu ändern

```
int main() {

    std::array<int, 3> arr = { 1, 2, 3 };
    // change all elements of the array to 100
    arr.fill(100);

}
```

std :: array online lesen: https://riptutorial.com/de/cplusplus/topic/2712/std----array

Kapitel 113: std :: atomics

Examples

Atomtypen

Jede Instantiierung und vollständige Spezialisierung der `std::atomic` Vorlage definiert einen atomaren Typ. Wenn ein Thread in ein atomares Objekt schreibt, während ein anderer Thread daraus liest, ist das Verhalten genau definiert (Informationen über Datenrennen finden Sie unter Speichermodell).

Darüber hinaus können Zugriffe auf atomare Objekte eine Inter-Thread-Synchronisation herstellen und nicht-atomare Speicherzugriffe `std::memory_order` wie in `std::memory_order` .

std :: atomic kann mit jedem `TriviallyCopyable type T. std::atomic` instanziiert werden `TriviallyCopyable type T. std::atomic` ist weder kopierbar noch beweglich.

Die Standardbibliothek bietet Spezialisierungen der Vorlage std :: atomic für die folgenden Typen:

1. Eine volle Spezialisierung für den Typ `bool` und seine typedef Name definiert , die als nicht-spezialisierten behandelt wird `std::atomic<T>` , außer dass es Standard - Layout hat, trivial Standardkonstruktors, trivial Destruktoren und unterstützt Aggregat Initialisierung Syntax:

Typedef Name	Volle Spezialisierung
std::atomic_bool	std::atomic<bool>

2) Vollständige Spezialisierungen und Typedefs für ganzzahlige Typen wie folgt:

Typedef Name	Volle Spezialisierung
std::atomic_char	std::atomic<char>
std::atomic_char	std::atomic<char>
std::atomic_schar	std::atomic<signed char>
std::atomic_uchar	std::atomic<unsigned char>
std::atomic_short	std::atomic<short>
std::atomic_ushort	std::atomic<unsigned short>
std::atomic_int	std::atomic<int>
std::atomic_uint	std::atomic<unsigned int>
std::atomic_long	std::atomic<long>
std::atomic_ulong	std::atomic<unsigned long>

Typedef Name	Volle Spezialisierung
std::atomic_llong	std::atomic<long long>
std::atomic_ullong	std::atomic<unsigned long long>
std::atomic_char16_t	std::atomic<char16_t>
std::atomic_char32_t	std::atomic<char32_t>
std::atomic_wchar_t	std::atomic<wchar_t>
std::atomic_int8_t	std::atomic<std::int8_t>
std::atomic_uint8_t	std::atomic<std::uint8_t>
std::atomic_int16_t	std::atomic<std::int16_t>
std::atomic_uint16_t	std::atomic<std::uint16_t>
std::atomic_int32_t	std::atomic<std::int32_t>
std::atomic_uint32_t	std::atomic<std::uint32_t>
std::atomic_int64_t	std::atomic<std::int64_t>
std::atomic_uint64_t	std::atomic<std::uint64_t>
std::atomic_int_least8_t	std::atomic<std::int_least8_t>
std::atomic_uint_least8_t	std::atomic<std::uint_least8_t>
std::atomic_int_least16_t	std::atomic<std::int_least16_t>
std::atomic_uint_least16_t	std::atomic<std::uint_least16_t>
std::atomic_int_least32_t	std::atomic<std::int_least32_t>
std::atomic_uint_least32_t	std::atomic<std::uint_least32_t>
std::atomic_int_least64_t	std::atomic<std::int_least64_t>
std::atomic_uint_least64_t	std::atomic<std::uint_least64_t>
std::atomic_int_fast8_t	std::atomic<std::int_fast8_t>
std::atomic_uint_fast8_t	std::atomic<std::uint_fast8_t>
std::atomic_int_fast16_t	std::atomic<std::int_fast16_t>
std::atomic_uint_fast16_t	std::atomic<std::uint_fast16_t>
std::atomic_int_fast32_t	std::atomic<std::int_fast32_t>
std::atomic_uint_fast32_t	std::atomic<std::uint_fast32_t>
std::atomic_int_fast64_t	std::atomic<std::int_fast64_t>
std::atomic_uint_fast64_t	std::atomic<std::uint_fast64_t>
std::atomic_intptr_t	std::atomic<std::intptr_t>
std::atomic_uintptr_t	std::atomic<std::uintptr_t>

Typedef Name	Volle Spezialisierung
std::atomic_size_t	std::atomic<std::size_t>
std::atomic_ptrdiff_t	std::atomic<std::ptrdiff_t>
std::atomic_intmax_t	std::atomic<std::intmax_t>
std::atomic_uintmax_t	std::atomic<std::uintmax_t>

Einfaches Beispiel für die Verwendung von std :: atomic_int

```
#include <iostream>       // std::cout
#include <atomic>         // std::atomic, std::memory_order_relaxed
#include <thread>         // std::thread

std::atomic_int foo (0);

void set_foo(int x) {
  foo.store(x,std::memory_order_relaxed);     // set value atomically
}

void print_foo() {
  int x;
  do {
    x = foo.load(std::memory_order_relaxed);  // get value atomically
  } while (x==0);
  std::cout << "foo: " << x << '\n';
}

int main ()
{
  std::thread first (print_foo);
  std::thread second (set_foo,10);
  first.join();
  //second.join();
  return 0;
}
//output: foo: 10
```

std :: atomics online lesen: https://riptutorial.com/de/cplusplus/topic/7475/std----atomics

Kapitel 114: std :: forward_list

Einführung

`std::forward_list` ist ein Container, der das schnelle Einfügen und Entfernen von Elementen von überall im Container unterstützt. Schneller Direktzugriff wird nicht unterstützt. Es ist als einfach verknüpfte Liste implementiert und hat im Wesentlichen keinen Overhead im Vergleich zu seiner Implementierung in C. Im Vergleich zu `std::list` bietet dieser Container mehr Speicherplatz, wenn keine bidirektionale Iteration erforderlich ist.

Bemerkungen

Durch das Hinzufügen, Entfernen und Verschieben der Elemente in der Liste oder über mehrere Listen hinweg werden die Iteratoren, die derzeit auf andere Elemente in der Liste verweisen, nicht ungültig. Ein Iterator oder eine Referenz, die sich auf ein Element bezieht, wird jedoch ungültig, wenn das entsprechende Element (über erase_after) aus der Liste entfernt wird. std :: forward_list erfüllt die Anforderungen von Container (mit Ausnahme der Größenelementfunktion und der Komplexität des Operators == ist immer linear), AllocatorAwareContainer und SequenceContainer.

Examples

Beispiel

```
#include <forward_list>
#include <string>
#include <iostream>

template<typename T>
std::ostream& operator<<(std::ostream& s, const std::forward_list<T>& v) {
    s.put('[');
    char comma[3] = {'\0', ' ', '\0'};
    for (const auto& e : v) {
        s << comma << e;
        comma[0] = ',';
    }
    return s << ']';
}

int main()
{
    // c++11 initializer list syntax:
    std::forward_list<std::string> words1 {"the", "frogurt", "is", "also", "cursed"};
    std::cout << "words1: " << words1 << '\n';

    // words2 == words1
    std::forward_list<std::string> words2(words1.begin(), words1.end());
    std::cout << "words2: " << words2 << '\n';

    // words3 == words1
```

```
    std::forward_list<std::string> words3(words1);
    std::cout << "words3: " << words3 << '\n';

    // words4 is {"Mo", "Mo", "Mo", "Mo", "Mo"}
    std::forward_list<std::string> words4(5, "Mo");
    std::cout << "words4: " << words4 << '\n';
}
```

Ausgabe:

```
words1: [the, frogurt, is, also, cursed]
words2: [the, frogurt, is, also, cursed]
words3: [the, frogurt, is, also, cursed]
words4: [Mo, Mo, Mo, Mo, Mo]
```

Methoden

Methodenname	Definition
`operator=`	weist dem Container Werte zu
`assign`	weist dem Container Werte zu
`get_allocator`	gibt den zugeordneten Allokator zurück
------	------
Elementzugriff	
`front`	greife auf das erste Element zu
------	------
Iteratoren	
`before_begi` n	Gibt einen Iterator an das Element zurück, bevor er beginnt
`cbefore_begin`	gibt einen konstanten Iterator an das Element zurück, bevor er beginnt
`begin`	bringt einen Iterator an den Anfang zurück
`cbegin`	bringt einen const-Iterator an den Anfang zurück
`end`	bringt einen Iterator zum Ende zurück
`cend`	bringt einen Iterator zum Ende zurück
Kapazität	
`empty`	prüft, ob der Container leer ist
`max_size`	gibt die maximal mögliche Anzahl von Elementen zurück

Methodenname	Definition
Modifikatoren	
`clear`	löscht den Inhalt
`insert_after`	fügt Elemente nach einem Element ein
`emplace_after`	Konstruiert Elemente direkt nach einem Element
`erase_after`	löscht ein Element nach einem Element
`push_front`	fügt ein Element an den Anfang ein
`emplace_front`	Konstruiert ein Element am Anfang
`pop_front`	Entfernt das erste Element
`resize`	ändert die Anzahl der gespeicherten Elemente
`swap`	tauscht den Inhalt aus
Operationen	
`merge`	führt zwei sortierte Listen zusammen
`splice_after`	verschiebt Elemente aus einer anderen forward_list
`remove`	entfernt Elemente, die bestimmte Kriterien erfüllen
`remove_if`	entfernt Elemente, die bestimmte Kriterien erfüllen
`reverse`	kehrt die Reihenfolge der Elemente um
`unique`	entfernt aufeinanderfolgende doppelte Elemente
`sort`	sortiert die Elemente

std :: forward_list online lesen: https://riptutorial.com/de/cplusplus/topic/9703/std----forward-list

Kapitel 115: std :: function: Um ein Element aufzurufen, das aufrufbar ist

Examples

Einfache Benutzung

```
#include <iostream>
#include <functional>
std::function<void(int , const std::string&)> myFuncObj;
void theFunc(int i, const std::string& s)
{
    std::cout << s << ": " << i << std::endl;
}
int main(int argc, char *argv[])
{
    myFuncObj = theFunc;
    myFuncObj(10, "hello world");
}
```

std :: function wird mit std :: bind verwendet

Denken Sie an eine Situation, in der wir eine Funktion mit Argumenten zurückrufen müssen. `std::function` mit `std::bind` verwendete `std::function` `std::bind` ergibt ein sehr leistungsfähiges Konstrukt wie unten gezeigt.

```
class A
{
public:
    std::function<void(int, const std::string&)> m_CbFunc = nullptr;
    void foo()
    {
        if (m_CbFunc)
        {
            m_CbFunc(100, "event fired");
        }
    }

};

class B
{
public:
    B()
    {
        auto aFunc = std::bind(&B::eventHandler, this, std::placeholders::_1,
std::placeholders::_2);
        anObjA.m_CbFunc = aFunc;
    }
    void eventHandler(int i, const std::string& s)
    {
        std::cout << s << ": " << i << std::endl;
```

```
    }

    void DoSomethingOnA()
    {
        anObjA.foo();
    }

    A anObjA;
};

int main(int argc, char *argv[])
{
     B anObjB;
     anObjB.DoSomethingOnA();
}
```

std :: function mit Lambda und std :: bind

```
#include <iostream>
#include <functional>

using std::placeholders::_1; // to be used in std::bind example

int stdf_foobar (int x, std::function<int(int)> moo)
{
    return x + moo(x); // std::function moo called
}

int foo (int x) { return 2+x; }

int foo_2 (int x, int y) { return 9*x + y; }

int main()
{
    int a = 2;

    /* Function pointers */
    std::cout << stdf_foobar(a, &foo) << std::endl; // 6 ( 2 + (2+2) )
    // can also be: stdf_foobar(2, foo)

    /* Lambda expressions */
    /* An unnamed closure from a lambda expression can be
     * stored in a std::function object:
     */
    int capture_value = 3;
    std::cout << stdf_foobar(a,
                             [capture_value](int param) -> int { return 7 + capture_value *
param; })
              << std::endl;
    // result: 15 ==  value + (7 * capture_value * value) == 2 + (7 + 3 * 2)

    /* std::bind expressions */
    /* The result of a std::bind expression can be passed.
     * For example by binding parameters to a function pointer call:
     */
    int b = stdf_foobar(a, std::bind(foo_2, _1, 3));
    std::cout << b << std::endl;
    // b == 23 == 2 + ( 9*2 + 3 )
    int c = stdf_foobar(a, std::bind(foo_2, 5, _1));
```

```
    std::cout << c << std::endl;
    // c == 49 == 2 + ( 9*5 + 2 )

    return 0;
}
```

`function` overhead

`std::function` kann erheblichen Overhead verursachen. Da `std::function` [Wertsemantik] [1] hat, muss sie die angegebene aufrufbare Funktion in sich selbst kopieren oder verschieben. Da jedoch Callables eines beliebigen Typs verwendet werden können, muss häufig Speicher dynamisch zugewiesen werden, um dies auszuführen.

Einige `function` verfügen über eine sogenannte "Small-Object-Optimization", bei der kleine Typen (wie Funktionszeiger, Member-Zeiger oder Funktionen mit sehr wenig Status) direkt im `function` gespeichert werden. Aber auch das funktioniert nur, wenn der Typ `noexcept` Move ist. Darüber hinaus erfordert der C ++ - Standard nicht, dass alle Implementierungen eine bereitstellen.

Folgendes berücksichtigen:

```
//Header file
using MyPredicate = std::function<bool(const MyValue &, const MyValue &)>;

void SortMyContainer(MyContainer &C, const MyPredicate &pred);

//Source file
void SortMyContainer(MyContainer &C, const MyPredicate &pred)
{
    std::sort(C.begin(), C.end(), pred);
}
```

Ein Vorlagenparameter wäre die bevorzugte Lösung für `SortMyContainer` , aber nehmen wir an, dass dies aus irgendeinem Grund nicht möglich oder wünschenswert ist. `SortMyContainer` muss `pred` hinter seinem eigenen Aufruf nicht speichern. `pred` kann jedoch durchaus Speicher zuweisen, wenn der ihm zugewiesene Funktionsumfang nicht trivial ist.

`function` weist Speicher zu, da etwas zum Kopieren / Verschieben benötigt wird; `function` übernimmt den Besitz des Aufrufbaren. `SortMyContainer` muss jedoch nicht das `SortMyContainer` *besitzen* . es bezieht sich nur darauf. Die `function` hier ist also ein Overkill; Es kann effizient sein, aber nicht.

Es gibt keinen Standardfunktionstyp für Bibliotheken, der nur auf eine aufrufbare Funktion verweist. Es muss also eine alternative Lösung gefunden werden, oder Sie können wählen, ob Sie mit dem Overhead leben möchten.

Die `function` hat auch keine wirksamen Mittel, um zu steuern, woher die Speicherzuordnungen für das Objekt stammen. Ja, es gibt Konstruktoren, die einen `allocator` benötigen, aber [viele Implementierungen implementieren sie nicht korrekt ... oder sogar *überhaupt*] [2].

C ++ 17

Die `function` , die einen `allocator` nicht mehr Teil des Typs. Daher gibt es keine Möglichkeit, die Zuordnung zu verwalten.

Das Aufrufen einer `function` ist auch langsamer als das direkte Aufrufen des Inhalts. Da jede `function` aufrufbare Objekte enthalten kann, muss der Aufruf über eine `function` indirekt sein. Der Aufwand der aufrufenden `function` liegt in der Reihenfolge eines virtuellen Funktionsaufrufs.

Bindet std :: function an andere aufrufbare Typen

```
/*
 * This example show some ways of using std::function to call
 *  a) C-like function
 *  b) class-member function
 *  c) operator()
 *  d) lambda function
 *
 * Function call can be made:
 *  a) with right arguments
 *  b) argumens with different order, types and count
 */
#include <iostream>
#include <functional>
#include <iostream>
#include <vector>

using std::cout;
using std::endl;
using namespace std::placeholders;



// simple function to be called
double foo_fn(int x, float y, double z)
{
  double res = x + y + z;
  std::cout << "foo_fn called with arguments: "
            << x << ", " << y << ", " << z
            << " result is : " << res
            << std::endl;
  return res;
}

// structure with member function to call
struct foo_struct
{
    // member function to call
    double foo_fn(int x, float y, double z)
    {
        double res = x + y + z;
        std::cout << "foo_struct::foo_fn called with arguments: "
                << x << ", " << y << ", " << z
                << " result is : " << res
                << std::endl;
        return res;
    }
    // this member function has different signature - but it can be used too
    // please not that argument order is changed too
    double foo_fn_4(int x, double z, float y, long xx)
```

```
    {
        double res = x + y + z + xx;
        std::cout << "foo_struct::foo_fn_4 called with arguments: "
                << x << ", " << z << ", " << y << ", " << xx
                << " result is : " << res
                << std::endl;
        return res;
    }
    // overloaded operator() makes whole object to be callable
    double operator()(int x, float y, double z)
    {
        double res = x + y + z;
        std::cout << "foo_struct::operator() called with arguments: "
                << x << ", " << y << ", " << z
                << " result is : " << res
                << std::endl;
        return res;
    }
};


int main(void)
{
  // typedefs
  using function_type = std::function<double(int, float, double)>;

  // foo_struct instance
  foo_struct fs;

  // here we will store all binded functions
  std::vector<function_type> bindings;

  // var #1 - you can use simple function
  function_type var1 = foo_fn;
  bindings.push_back(var1);

  // var #2 - you can use member function
  function_type var2 = std::bind(&foo_struct::foo_fn, fs, _1, _2, _3);
  bindings.push_back(var2);

  // var #3 - you can use member function with different signature
  // foo_fn_4 has different count of arguments and types
  function_type var3 = std::bind(&foo_struct::foo_fn_4, fs, _1, _3, _2, 0l);
  bindings.push_back(var3);

  // var #4 - you can use object with overloaded operator()
  function_type var4 = fs;
  bindings.push_back(var4);

  // var #5 - you can use lambda function
  function_type var5 = [](int x, float y, double z)
    {
        double res = x + y + z;
        std::cout << "lambda  called with arguments: "
                << x << ", " << y << ", " << z
                << " result is : " << res
                << std::endl;
        return res;
    };
  bindings.push_back(var5);
```

```
  std::cout << "Test stored functions with arguments: x = 1, y = 2, z = 3"
            << std::endl;

  for (auto f : bindings)
      f(1, 2, 3);

}
```

Leben

Ausgabe:

```
Test stored functions with arguments: x = 1, y = 2, z = 3
foo_fn called with arguments: 1, 2, 3 result is : 6
foo_struct::foo_fn called with arguments: 1, 2, 3 result is : 6
foo_struct::foo_fn_4 called with arguments: 1, 3, 2, 0 result is : 6
foo_struct::operator() called with arguments: 1, 2, 3 result is : 6
lambda  called with arguments: 1, 2, 3 result is : 6
```

Speichern von Funktionsargumenten in std :: tuple

Einige Programme müssen daher Argumente für den späteren Aufruf einer Funktion speichern.

In diesem Beispiel wird gezeigt, wie eine Funktion mit in std :: tuple gespeicherten Argumenten aufgerufen wird

```
#include <iostream>
#include <functional>
#include <tuple>
#include <iostream>

// simple function to be called
double foo_fn(int x, float y, double z)
{
   double res =  x + y + z;
   std::cout << "foo_fn called. x = " << x << " y = " << y << " z = " << z
             << " res=" << res;
   return res;
}

// helpers for tuple unrolling
template<int ...> struct seq {};
template<int N, int ...S> struct gens : gens<N-1, N-1, S...> {};
template<int ...S> struct gens<0, S...>{ typedef seq<S...> type; };

// invocation helper
template<typename FN, typename P, int ...S>
double call_fn_internal(const FN& fn, const P& params, const seq<S...>)
{
   return fn(std::get<S>(params) ...);
}
// call function with arguments stored in std::tuple
template<typename Ret, typename ...Args>
Ret call_fn(const std::function<Ret(Args...)>& fn,
            const std::tuple<Args...>& params)
{
    return call_fn_internal(fn, params, typename gens<sizeof...(Args)>::type());
```

```
}


int main(void)
{
  // arguments
  std::tuple<int, float, double> t = std::make_tuple(1, 5, 10);
  // function to call
  std::function<double(int, float, double)> fn = foo_fn;

  // invoke a function with stored arguments
  call_fn(fn, t);
}
```

Leben

Ausgabe:

```
foo_fn called. x = 1 y = 5 z = 10 res=16
```

std :: function: Um ein Element aufzurufen, das aufrufbar ist online lesen: https://riptutorial.com/de/cplusplus/topic/2294/std----function--um-ein-element-aufzurufen--das-aufrufbar-ist

Kapitel 116: std :: integer_sequence

Einführung

Die Klassenvorlage `std::integer_sequence<Type, Values...>` repräsentiert eine Folge von Werten des Typs `Type` wobei `Type` einer der integrierten Integer-Typen ist. Diese Sequenzen werden bei der Implementierung von Klassen- oder Funktionsvorlagen verwendet, die vom Positionszugriff profitieren. Die Standardbibliothek enthält auch "Factory" -Typen, die aufsteigende Folgen von ganzzahligen Werten nur aus der Anzahl der Elemente erstellen.

Examples

Drehen Sie ein std :: Tupel in Funktionsparameter

Ein `std::tuple<T...>` kann verwendet werden, um mehrere Werte zu übergeben. Es könnte beispielsweise verwendet werden, um eine Folge von Parametern in einer Warteschlange zu speichern. Bei der Verarbeitung eines solchen Tupels müssen seine Elemente in Funktionsaufrufargumente umgewandelt werden:

```
#include <array>
#include <iostream>
#include <string>
#include <tuple>
#include <utility>

// ----------------------------------------------------------------------------
// Example functions to be called:
void f(int i, std::string const& s) {
    std::cout << "f(" << i << ", " << s << ")\n";
}
void f(int i, double d, std::string const& s) {
    std::cout << "f(" << i << ", " << d << ", " << s << ")\n";
}
void f(char c, int i, double d, std::string const& s) {
    std::cout << "f(" << c << ", " << i << ", " << d << ", " << s << ")\n";
}
void f(int i, int j, int k) {
    std::cout << "f(" << i << ", " << j << ", " << k << ")\n";
}

// ----------------------------------------------------------------------------
// The actual function expanding the tuple:
template <typename Tuple, std::size_t... I>
void process(Tuple const& tuple, std::index_sequence<I...>) {
    f(std::get<I>(tuple)...);
}

// The interface to call. Sadly, it needs to dispatch to another function
// to deduce the sequence of indices created from std::make_index_sequence<N>
template <typename Tuple>
void process(Tuple const& tuple) {
    process(tuple, std::make_index_sequence<std::tuple_size<Tuple>::value>());
```

```
}

// ------------------------------------------------------------------------
int main() {
    process(std::make_tuple(1, 3.14, std::string("foo")));
    process(std::make_tuple('a', 2, 2.71, std::string("bar")));
    process(std::make_pair(3, std::string("pair")));
    process(std::array<int, 3>{ 1, 2, 3 });
}
```

Solange eine Klasse `std::get<I>(object)` und `std::tuple_size<T>::value` , kann sie mit der obigen Funktion `process()` werden. Die Funktion selbst ist völlig unabhängig von der Anzahl der Argumente.

Erstellen Sie ein Parameterpaket, das aus Ganzzahlen besteht

`std::integer_sequence` selbst geht es um das Halten einer Folge von Ganzzahlen, die in ein Parameterpaket umgewandelt werden können. Sein primärer Wert ist die Möglichkeit, "Factory" Klassenvorlagen zu erstellen, die diese Sequenzen erstellen:

```
#include <iostream>
#include <initializer_list>
#include <utility>

template <typename T, T... I>
void print_sequence(std::integer_sequence<T, I...>) {
    std::initializer_list<bool>{ bool(std::cout << I << ' ')... };
    std::cout << '\n';
}

template <int Offset, typename T, T... I>
void print_offset_sequence(std::integer_sequence<T, I...>) {
    print_sequence(std::integer_sequence<T, T(I + Offset)...>());
}

int main() {
    // explicitly specify sequences:
    print_sequence(std::integer_sequence<int, 1, 2, 3>());
    print_sequence(std::integer_sequence<char, 'f', 'o', 'o'>());

    // generate sequences:
    print_sequence(std::make_index_sequence<10>());
    print_sequence(std::make_integer_sequence<short, 10>());
    print_offset_sequence<'A'>(std::make_integer_sequence<char, 26>());
}
```

Die Funktionsvorlage `print_sequence()` verwendet beim Erweitern der Ganzzahlsequenz eine `std::initializer_list<bool>` , um die Reihenfolge der Auswertung zu gewährleisten und keine ungenutzte [array] `print_sequence()` erstellen.

Verwandeln Sie eine Folge von Indizes in Kopien eines Elements

Durch Erweitern des Parameterpakets von Indizes in einem Kommaausdruck um einen Wert wird eine Kopie des Werts für jeden der Indizes erstellt. Leider glauben `gcc` und `clang` dass der Index

keine Auswirkung hat und warnen (`gcc` kann zum Schweigen gebracht werden, indem der Index für `void`):

```
#include <algorithm>
#include <array>
#include <iostream>
#include <iterator>
#include <string>
#include <utility>

template <typename T, std::size_t... I>
std::array<T, sizeof...(I)> make_array(T const& value, std::index_sequence<I...>) {
    return std::array<T, sizeof...(I)>{ (I, value)... };
}

template <int N, typename T>
std::array<T, N> make_array(T const& value) {
    return make_array(value, std::make_index_sequence<N>());
}

int main() {
    auto array = make_array<20>(std::string("value"));
    std::copy(array.begin(), array.end(),
              std::ostream_iterator<std::string>(std::cout, " "));
    std::cout << "\n";
}
```

std :: integer_sequence online lesen: https://riptutorial.com/de/cplusplus/topic/8315/std----integer-sequence

Kapitel 117: std :: iomanip

Examples

std :: setw

```
int val = 10;
// val will be printed to the extreme left end of the  output console:
std::cout << val << std::endl;
 // val will be printed in an output field of length 10 starting from right end of the field:
std::cout << std::setw(10) << val << std::endl;
```

Dies gibt aus:

```
10
        10
1234567890
```

(wo die letzte Zeile dazu dient, die Zeichenversätze zu sehen).

Manchmal müssen wir die Breite des Ausgabefeldes einstellen, normalerweise, wenn wir die Ausgabe in einem strukturierten und richtigen Layout erhalten möchten. **`std::setw`** kann mit **`std::setw`** von **std :: iomanip durchgeführt werden** .

Die Syntax für `std::setw` lautet:

```
std::setw(int n)
```

Dabei ist n die Länge des einzustellenden Ausgabefeldes

std :: setprecision

Bei Verwendung in einem Ausdruck `out << setprecision(n)` oder `in >> setprecision(n)` wird der Genauigkeitsparameter des Streams auf n genau gesetzt. Parameter dieser Funktion ist Integer, was für die Genauigkeit ein neuer Wert ist.

Beispiel:

```
#include <iostream>
#include <iomanip>
#include <cmath>
#include <limits>
int main()
{
    const long double pi = std::acos(-1.L);
    std::cout << "default precision (6): " << pi << '\n'
              << "std::precision(10):    " << std::setprecision(10) << pi << '\n'
              << "max precision:         "
              << std::setprecision(std::numeric_limits<long double>::digits10 + 1)
```

```
              << pi << '\n';
}
//Output
//default precision (6): 3.14159
//std::precision(10):    3.141592654
//max precision:         3.141592653589793239
```

std :: setfill

Bei Verwendung in einem Ausdruck `out << setfill(c)` das Füllzeichen des Streams auf `c` .

Hinweis: Das aktuelle Füllzeichen kann mit `std::ostream::fill` .

Beispiel:

```
#include <iostream>
#include <iomanip>
int main()
{
    std::cout << "default fill: " << std::setw(10) << 42 << '\n'
          << "setfill('*'): " << std::setfill('*')
                              << std::setw(10) << 42 << '\n';
}
//output::
//default fill:         42
//setfill('*'): ********42
```

std :: setiosflags

Bei Verwendung in einem Ausdruck `out << setiosflags(mask)` oder `in >> setiosflags(mask)` werden alle Format-Flags des Streams `in >> setiosflags(mask)` oder `in >> setiosflags(mask)` , wie in der Maske angegeben.

Liste aller `std::ios_base::fmtflags` :

- `dec` - Dezimalbasis für ganzzahlige E / A verwenden
- `oct` - Oktalbasis für ganzzahlige E / A verwenden
- `hex` - verwendet eine hexadezimale Basis für ganzzahlige E / A
- `basefield` - `dec|oct|hex|0` nützlich für Maskierungsoperationen
- `left` Links-Einstellung (Füllzeichen rechts hinzufügen)
- `right` Rechts-Einstellung (füllt links Füllzeichen)
- `internal` - interne Anpassung (fügt Füllzeichen zum internen festgelegten Punkt hinzu)
- `adjustfield` - `left|right|internal` . Nützlich für Maskierungsvorgänge
- `scientific` - Generieren Sie Fließkomma-Typen mit Hilfe der wissenschaftlichen Notation oder der Hex-Notation, wenn Sie sie mit einer festen Schreibweise kombinieren
- `fixed` - Generiert Fließkommatypen mit fester Notation oder Hex-Notation in Kombination mit wissenschaftlicher
- `floatfield` - `scientific|fixed|(scientific|fixed)|0` . Nützlich für Maskierungsvorgänge
- `boolalpha` - Einfügen und Extrahieren des `bool` Typs im alphanumerischen Format
- `showbase` - Generieren Sie ein Präfix, das die numerische Basis für die Ganzzahlausgabe angibt, und fordern Sie das Währungskennzeichen in monetärer E / A an

- `showpoint` - Erzeugt uneingeschränkt ein Dezimalzeichen für die Ausgabe von Gleitkommazahlen
- `showpos` - ein generieren `+` Zeichen für nicht-negative numerische Ausgabe
- `skipws` - Überspringt führende Leerzeichen vor bestimmten Eingabevorgängen
- `unitbuf` den Ausgang nach jeder Ausgabeoperation
- `uppercase` - Ersetzen Sie bestimmte Kleinbuchstaben in bestimmten Ausgabeoperationen durch Großbuchstaben

Beispiel für Manipulatoren:

```
#include <iostream>
#include <string>
#include<iomanip>
int main()
{
  int l_iTemp = 47;
  std::cout<<  std::resetiosflags(std::ios_base::basefield);
  std::cout<<std::setiosflags( std::ios_base::oct)<<l_iTemp<<std::endl;
  //output: 57
  std::cout<<  std::resetiosflags(std::ios_base::basefield);
  std::cout<<std::setiosflags( std::ios_base::hex)<<l_iTemp<<std::endl;
  //output: 2f
  std::cout<<std::setiosflags( std::ios_base::uppercase)<<l_iTemp<<std::endl;
  //output 2F
  std::cout<<std::setfill('0')<<std::setw(12);
  std::cout<<std::resetiosflags(std::ios_base::uppercase);
  std::cout<<std::setiosflags( std::ios_base::right)<<l_iTemp<<std::endl;
  //output: 00000000002f

  std::cout<<std::resetiosflags(std::ios_base::basefield|std::ios_base::adjustfield);
  std::cout<<std::setfill('.')<<std::setw(10);
  std::cout<<std::setiosflags( std::ios_base::left)<<l_iTemp<<std::endl;
  //output: 47........

  std::cout<<std::resetiosflags(std::ios_base::adjustfield)<<std::setfill('#');
  std::cout<<std::setiosflags(std::ios_base::internal|std::ios_base::showpos);
  std::cout<<std::setw(10)<<l_iTemp<<std::endl;
  //output +########47

  double l_dTemp = -1.2;
  double pi = 3.14159265359;
  std::cout<<pi<<"    "<<l_dTemp<<std::endl;
  //output +3.14159   -1.2
  std::cout<<std::setiosflags(std::ios_base::showpoint)<<l_dTemp<<std::endl;
  //output -1.20000
  std::cout<<setiosflags(std::ios_base::scientific)<<pi<<std::endl;
  //output: +3.141593e+00
  std::cout<<std::resetiosflags(std::ios_base::floatfield);
  std::cout<<setiosflags(std::ios_base::fixed)<<pi<<std::endl;
  //output: +3.141593
  bool b = true;
  std::cout<<std::setiosflags(std::ios_base::unitbuf|std::ios_base::boolalpha)<<b;
  //output: true
  return 0;
}
```

std :: iomanip online lesen: https://riptutorial.com/de/cplusplus/topic/6936/std----iomanip

Kapitel 118: std :: map

Bemerkungen

- Um `std::map` oder `std::multimap` die Header-Datei `<map>` enthalten sein.
- Sowohl `std::map` als auch `std::multimap` sortieren ihre Elemente nach der aufsteigenden Reihenfolge der Schlüssel. Bei `std::multimap` erfolgt keine Sortierung für die Werte desselben Schlüssels.
- Der grundlegende Unterschied zwischen `std::map` und `std::multimap` besteht darin, dass der `std::map` `std::multimap` keine doppelten Werte für denselben Schlüssel zulässt, für den `std::multimap` gilt.
- Karten werden als binäre Suchbäume implementiert. `search()` , `insert()` , `erase()` dauert also durchschnittlich Θ (log n). Verwenden Sie für konstante Zeit den `std::unordered_map` .
- `size()` und `empty()` haben eine time (1) -Komplexität. Die Anzahl der Knoten wird zwischengespeichert, um zu vermeiden, dass bei jedem Aufruf dieser Funktionen der Baum durchlaufen wird.

Examples

Zugriff auf Elemente

Eine `std::map (key, value)` Paare als Eingabe.

Betrachten Sie das folgende Beispiel für die Initialisierung von `std::map` :

```
std::map < std::string, int > ranking { std::make_pair("stackoverflow", 2),
                                        std::make_pair("docs-beta", 1) };
```

In einer `std::map` können Elemente wie folgt eingefügt werden:

```
ranking["stackoverflow"]=2;
ranking["docs-beta"]=1;
```

In dem obigen Beispiel, wenn der Schlüssel `stackoverflow` bereits vorhanden ist, wird sein Wert auf 2 aktualisiert werden , wenn es nicht bereits vorhanden ist , wird ein neuer Eintrag erstellt werden soll.

In einer `std::map` kann auf Elemente direkt zugegriffen werden, indem der Schlüssel als Index angegeben wird:

```
std::cout << ranking[ "stackoverflow" ] << std::endl;
```

Beachten Sie, dass bei Verwendung des `operator[]` auf der Karte tatsächlich *ein neuer Wert* mit dem abgefragten Schlüssel in die Karte eingefügt wird. Das bedeutet, dass Sie es nicht auf einer `const std::map` , selbst wenn der Schlüssel bereits in der Map gespeichert ist. Um diese Einfügung zu verhindern, prüfen Sie, ob das Element vorhanden ist (beispielsweise mit `find()`) oder verwenden Sie `at()` wie unten beschrieben.

C ++ 11

Auf Elemente einer `std::map` kann mit `at()` zugegriffen werden:

```
std::cout << ranking.at("stackoverflow") << std::endl;
```

Beachten Sie, dass `at()` eine Ausnahme von `std::out_of_range` wenn der Container das angeforderte Element nicht enthält.

In beiden Containern `std::map` und `std::multimap` kann auf Elemente mit Iteratoren zugegriffen werden:

C ++ 11

```
// Example using begin()
std::multimap < int, std::string > mmp { std::make_pair(2, "stackoverflow"),
                                         std::make_pair(1, "docs-beta"),
                                         std::make_pair(2, "stackexchange")  };
auto it = mmp.begin();
std::cout << it->first << " : " << it->second << std::endl; // Output: "1 : docs-beta"
it++;
std::cout << it->first << " : " << it->second << std::endl; // Output: "2 : stackoverflow"
it++;
std::cout << it->first << " : " << it->second << std::endl; // Output: "2 : stackexchange"

// Example using rbegin()
std::map < int, std::string > mp {  std::make_pair(2, "stackoverflow"),
                                    std::make_pair(1, "docs-beta"),
                                    std::make_pair(2, "stackexchange")  };
auto it2 = mp.rbegin();
std::cout << it2->first << " : " << it2->second << std::endl; // Output: "2 : stackoverflow"
it2++;
std::cout << it2->first << " : " << it2->second << std::endl; // Output: "1 : docs-beta"
```

Std :: map oder std :: multimap initialisieren

`std::map` und `std::multimap` beide durch Angabe von durch Komma getrennten Schlüssel-Wert-Paaren initialisiert werden. Schlüssel-Wert-Paare können entweder von `{key, value}` bereitgestellt werden oder explizit von `std::make_pair(key, value)` . Da `std::map` keine doppelten Schlüssel zulässt und der Kommaoperator von rechts nach links ausgeführt wird, wird das rechte Paar mit dem gleichen Schlüssel links überschrieben.

```
std::multimap < int, std::string > mmp { std::make_pair(2, "stackoverflow"),
                                         std::make_pair(1, "docs-beta"),
                                         std::make_pair(2, "stackexchange")  };
// 1 docs-beta
// 2 stackoverflow
```

```
// 2 stackexchange

std::map < int, std::string > mp {  std::make_pair(2, "stackoverflow"),
                                    std::make_pair(1, "docs-beta"),
                                    std::make_pair(2, "stackexchange")  };
// 1 docs-beta
// 2 stackoverflow
```

Beide konnten mit Iterator initialisiert werden.

```
// From std::map or std::multimap iterator
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {6, 8}, {3, 4},
                                {6, 7} };
                        // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 8}, {6, 7}, {8, 9}
auto it = mmp.begin();
std::advance(it,3); //moved cursor on first {6, 5}
std::map< int, int > mp(it, mmp.end()); // {6, 5}, {8, 9}

//From std::pair array
std::pair< int, int > arr[10];
arr[0] = {1, 3};
arr[1] = {1, 5};
arr[2] = {2, 5};
arr[3] = {0, 1};
std::map< int, int > mp(arr,arr+4); //{0 , 1}, {1, 3}, {2, 5}

//From std::vector of std::pair
std::vector< std::pair<int, int> > v{ {1, 5}, {5, 1}, {3, 6}, {3, 2} };
std::multimap< int, int > mp(v.begin(), v.end());
                         // {1, 5}, {3, 6}, {3, 2}, {5, 1}
```

Elemente löschen

Alle Elemente entfernen:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
mmp.clear(); //empty multimap
```

Element mit Hilfe des Iterators irgendwo entfernen:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
                             // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
auto it = mmp.begin();
std::advance(it,3); // moved cursor on first {6, 5}
mmp.erase(it); // {1, 2}, {3, 4}, {3, 4}, {6, 7}, {8, 9}
```

Alle Elemente eines Bereichs entfernen:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
                             // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
auto it = mmp.begin();
auto it2 = it;
it++; //moved first cursor on first {3, 4}
std::advance(it2,3);  //moved second cursor on first {6, 5}
mmp.erase(it,it2); // {1, 2}, {6, 5}, {6, 7}, {8, 9}
```

Entfernen aller Elemente mit einem angegebenen Wert als Schlüssel:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
                            // {1, 2}, {3, 4}, {3, 4}, {6, 5}, {6, 7}, {8, 9}
mmp.erase(6); // {1, 2}, {3, 4}, {3, 4}, {8, 9}
```

Entfernen von Elementen , die ein Prädikat erfüllen `pred` :

```
std::map<int,int> m;
auto it = m.begin();
while (it != m.end())
{
   if (pred(*it))
       it = m.erase(it);
   else
       ++it;
}
```

Elemente einfügen

Ein Element kann nur dann in eine `std::map` eingefügt werden, wenn sein Schlüssel nicht bereits in der Map vorhanden ist. Zum Beispiel gegeben:

```
std::map< std::string, size_t > fruits_count;
```

- Ein Schlüsselwertpaar wird über die `insert()` in eine `std::map` eingefügt. Es erfordert ein `pair` als Argument:

  ```
  fruits_count.insert({"grapes", 20});
  fruits_count.insert(make_pair("orange", 30));
  fruits_count.insert(pair<std::string, size_t>("banana", 40));
  fruits_count.insert(map<std::string, size_t>::value_type("cherry", 50));
  ```

  Die Funktion `insert()` gibt ein `pair` das aus einem Iterator und einem `bool` Wert besteht:

  - Wenn das Einfügen erfolgreich war, zeigt der Iterator auf das neu eingefügte Element, und der `bool` Wert ist `true` .
  - Wenn bereits ein Element mit demselben `key` , schlägt das Einfügen fehl. In diesem Fall zeigt der Iterator auf das Element, das den Konflikt verursacht, und der `bool` Wert ist " `false` .

  Die folgende Methode kann zum Kombinieren von Einfügungs- und Suchvorgängen verwendet werden:

  ```
  auto success = fruits_count.insert({"grapes", 20});
  if (!success.second) {           // we already have 'grapes' in the map
      success.first->second += 20; // access the iterator to update the value
  }
  ```

- Der Container `std::map` bietet dem `std::map` die Möglichkeit, auf Elemente in der Map zuzugreifen und neue Elemente einzufügen, falls diese nicht vorhanden sind:

```
fruits_count["apple"] = 10;
```

Dies verhindert zwar, dass der Benutzer bereits überprüft, ob das Element bereits vorhanden ist. Wenn ein Element fehlt, erstellt `std::map::operator[]` implizit und initialisiert es mit dem Standardkonstruktor, bevor es mit dem angegebenen Wert überschrieben wird.

- `insert()` können Sie mehrere Elemente auf einmal mit einer geschweiften Liste von Paaren hinzufügen. Diese Version von insert () gibt void zurück:

```
fruits_count.insert({{"apricot", 1}, {"jackfruit", 1}, {"lime", 1}, {"mango", 7}});
```

- `insert()` kann auch zum Hinzufügen von Elementen verwendet werden, indem Iteratoren verwendet werden, die den Anfang und das Ende der Werte von `value_type` :

```
std::map< std::string, size_t > fruit_list{ {"lemon", 0}, {"olive", 0}, {"plum", 0}};
fruits_count.insert(fruit_list.begin(), fruit_list.end());
```

Beispiel:

```
std::map<std::string, size_t> fruits_count;
std::string fruit;
while(std::cin >> fruit){
    // insert an element with 'fruit' as key and '1' as value
    // (if the key is already stored in fruits_count, insert does nothing)
    auto ret = fruits_count.insert({fruit, 1});
    if(!ret.second){            // 'fruit' is already in the map
        ++ret.first->second;    // increment the counter
    }
}
```

Die Zeitkomplexität für eine Einfügeoperation ist O (log n), da `std::map` als Bäume implementiert ist.

C ++ 11

Ein `pair` kann explizit mit `make_pair()` und `emplace()` :

```
std::map< std::string , int > runs;
runs.emplace("Babe Ruth", 714);
runs.insert(make_pair("Barry Bonds", 762));
```

Wenn wir wissen, wo das neue Element eingefügt wird, können Sie mit `emplace_hint()` einen Iterator- `hint` angeben. Wenn das neue Element unmittelbar vor dem `hint` eingefügt werden kann, kann das Einfügen in konstanter Zeit erfolgen. Ansonsten verhält es sich genauso wie `emplace()` :

```
std::map< std::string , int > runs;
auto it = runs.emplace("Barry Bonds", 762); // get iterator to the inserted element
// the next element will be before "Barry Bonds", so it is inserted before 'it'
runs.emplace_hint(it, "Babe Ruth", 714);
```

Iteration über std :: map oder std :: multimap

`std::map` oder `std::multimap` kann auf folgende Weise durchlaufen werden:

```
std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };

//Range based loop - since C++11
for(const auto &x: mmp)
    std::cout<< x.first <<":"<< x.second << std::endl;

//Forward iterator for loop: it would loop through first element to last element
//it will be a std::map< int, int >::iterator
for (auto it = mmp.begin(); it != mmp.end(); ++it)
std::cout<< it->first <<":"<< it->second << std::endl; //Do something with iterator

//Backward iterator for loop: it would loop through last element to first element
//it will be a std::map< int, int >::reverse_iterator
for (auto it = mmp.rbegin(); it != mmp.rend(); ++it)
std::cout<< it->first <<" "<< it->second << std::endl; //Do something with iterator
```

Während über einen Iterieren `std::map` oder `std::multimap` , die Verwendung von `auto` bevorzugt nutzlos implizite Konvertierungen zu vermeiden (siehe diese SO Antwort für weitere Details).

Suchen in std :: map oder in std :: multimap

Es gibt mehrere Möglichkeiten, einen Schlüssel in `std::map` oder in `std::multimap` zu suchen.

- Um den Iterator des ersten Vorkommens eines Schlüssels zu erhalten, kann die Funktion `find()` verwendet werden. Es gibt `end()` wenn der Schlüssel nicht vorhanden ist.

  ```
  std::multimap< int , int > mmp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
  auto it = mmp.find(6);
  if(it!=mmp.end())
      std::cout << it->first << ", " << it->second << std::endl; //prints: 6, 5
  else
      std::cout << "Value does not exist!" << std::endl;

  it = mmp.find(66);
  if(it!=mmp.end())
      std::cout << it->first << ", " << it->second << std::endl;
  else
      std::cout << "Value does not exist!" << std::endl; // This line would be executed.
  ```

- Eine andere Möglichkeit, um herauszufinden, ob ein Eintrag in `std::map` oder in `std::multimap` ist, ist die Funktion `count()` , die zählt, wie viele Werte einem bestimmten Schlüssel zugeordnet sind. Da `std::map` jedem Schlüssel nur einen Wert zuordnet, kann die Funktion `count()` nur 0 (wenn der Schlüssel nicht vorhanden ist) oder 1 (falls vorhanden) zurückgeben. Für `std::multimap` kann `count()` Werte größer als 1 zurückgeben, da dem gleichen Schlüssel mehrere Werte zugeordnet werden können.

  ```
  std::map< int , int > mp{ {1, 2}, {3, 4}, {6, 5}, {8, 9}, {3, 4}, {6, 7} };
  if(mp.count(3) > 0) // 3 exists as a key in map
      std::cout << "The key exists!" << std::endl; // This line would be executed.
  ```

```
else
    std::cout << "The key does not exist!" << std::endl;
```

Wenn Sie nur darauf achten, ob ein Element vorhanden ist, ist `find` strikt besser: Es dokumentiert Ihre Absicht und kann bei `multimaps` aufhören, sobald das erste passende Element gefunden wurde.

- Im Fall von `std::multimap` könnten mehrere Elemente den gleichen Schlüssel haben. Um diesen Bereich zu erhalten, wird die Funktion `equal_range()` verwendet, die `std::pair` mit Iterator-Untergrenze (einschließlich) und Obergrenze (exklusiv) zurückgibt. Wenn der Schlüssel nicht vorhanden ist, zeigen beide Iteratoren auf `end()` .

```
auto eqr = mmp.equal_range(6);
auto st = eqr.first, en = eqr.second;
for(auto it = st; it != en; ++it){
    std::cout << it->first << ", " << it->second << std::endl;
}
    // prints: 6, 5
    //         6, 7
```

Anzahl der Elemente prüfen

Der Container `std::map` hat eine Memberfunktion `empty()` , die `true` oder `false` zurückgibt, je nachdem, ob die Map leer ist oder nicht. Die Elementfunktion `size()` gibt die Anzahl der Elemente zurück, die in einem `std::map` Container gespeichert sind:

```
std::map<std::string , int> rank {{"facebook.com", 1} ,{"google.com", 2}, {"youtube.com", 3}};
if(!rank.empty()){
    std::cout << "Number of elements in the rank map: " << rank.size() << std::endl;
}
else{
    std::cout << "The rank map is empty" << std::endl;
}
```

Arten von Karten

Regelmäßige Karte

Eine Map ist ein assoziativer Container, der Schlüsselwertpaare enthält.

```
#include <string>
#include <map>
std::map<std::string, size_t> fruits_count;
```

Im obigen Beispiel ist `std::string` der *Schlüsseltyp* und `size_t` ist ein *Wert* .

Der Schlüssel dient als Index in der Karte. Jeder Schlüssel muss eindeutig sein und muss bestellt werden.

- Wenn Sie mehrere Elemente mit demselben Schlüssel benötigen, ziehen Sie die

Verwendung von `multimap` Betracht (unten erklärt).

- Wenn Ihr Werttyp keine Reihenfolge angibt oder Sie die Standardreihenfolge überschreiben möchten, können Sie eine angeben:

```
#include <string>
#include <map>
#include <cstring>
struct StrLess {
    bool operator()(const std::string& a, const std::string& b) {
        return strncmp(a.c_str(), b.c_str(), 8)<0;
               //compare only up to 8 first characters
    }
}
std::map<std::string, size_t, StrLess> fruits_count2;
```

Wenn der `StrLess` Komparator für zwei Schlüssel den `StrLess` `false` zurückgibt, werden sie als identisch betrachtet, auch wenn der tatsächliche Inhalt unterschiedlich ist.

Multi-Map

Mit Multimap können mehrere Schlüsselwertpaare mit demselben Schlüssel in der Karte gespeichert werden. Ansonsten ist die Benutzeroberfläche und Erstellung der normalen Map sehr ähnlich.

```
#include <string>
#include <map>
std::multimap<std::string, size_t> fruits_count;
std::multimap<std::string, size_t, StrLess> fruits_count2;
```

Hash-Map (ungeordnete Karte)

Eine Hash-Map speichert Schlüsselwertpaare ähnlich einer regulären Map. Es ordnet die Elemente jedoch nicht in Bezug auf den Schlüssel an. Stattdessen wird ein Hashwert für den Schlüssel verwendet, um schnell auf die erforderlichen Schlüssel-Wert-Paare zuzugreifen.

```
#include <string>
#include <unordered_map>
std::unordered_map<std::string, size_t> fruits_count;
```

Ungeordnete Karten sind normalerweise schneller, aber die Elemente werden nicht in einer vorhersagbaren Reihenfolge gespeichert. Wenn Sie beispielsweise alle Elemente in einer `unordered_map` durchlaufen, werden die Elemente in einer scheinbar zufälligen Reihenfolge angezeigt.

Erstellen von std :: map mit benutzerdefinierten Typen als Schlüssel

Um eine Klasse als Schlüssel in einer Map verwenden zu können, muss der Schlüssel `copiable` und `assignable` . Die Reihenfolge innerhalb der Map wird durch das dritte Argument der Vorlage

(und das Argument des Konstruktors, falls verwendet) definiert. Der *Standardwert* ist `std::less<KeyType>` , der standardmäßig auf den `<` Operator, aber es gibt keine Notwendigkeit , die Standardeinstellung zu verwenden. Schreiben Sie einfach einen Vergleichsoperator (vorzugsweise als Funktionsobjekt):

```
struct CmpMyType
{
    bool operator()( MyType const& lhs, MyType const& rhs ) const
    {
        //  ...
    }
};
```

In C ++ muss das Vergleichselement "Compare" eine strikte schwache Reihenfolge sein . Insbesondere muss der `compare(X,X)` für jedes `X false` . dh wenn `CmpMyType()(a, b)` true zurückgibt, muss `CmpMyType()(b, a)` false zurückgeben, und wenn beide false zurückgeben, werden die Elemente als gleich betrachtet (Mitglieder derselben Äquivalenzklasse).

Strikte schwache Reihenfolge

Dies ist ein mathematischer Begriff, um eine Beziehung zwischen zwei Objekten zu definieren. Seine Definition ist:

> Zwei Objekte x und y sind gleichwertig, wenn sowohl f (x, y) als auch f (y, x) falsch sind. Beachten Sie, dass ein Objekt (durch die Irreflexivitätsinvariante) immer sich selbst entspricht.

In C ++ bedeutet dies, wenn Sie zwei Objekte eines bestimmten Typs haben, sollten Sie die folgenden Werte zurückgeben, wenn Sie sie mit dem Operator <vergleichen.

```
X    a;
X    b;

Condition:                  Test:     Result
a is equivalent to b:       a < b     false
a is equivalent to b        b < a     false

a is less than b            a < b     true
a is less than b            b < a     false

b is less than a            a < b     false
b is less than a            b < a     true
```

Wie Sie Äquivalent / Weniger definieren, hängt völlig vom Typ Ihres Objekts ab.

std :: map online lesen: https://riptutorial.com/de/cplusplus/topic/681/std----map

Kapitel 119: std :: optional

Examples

Einführung

Optionale (auch als Vielleicht-Typen bezeichnet) werden verwendet, um einen Typ darzustellen, dessen Inhalt möglicherweise vorhanden ist oder nicht. Sie sind in C ++ 17 als `std::optional` Klasse `std::optional` implementiert. Ein Objekt des Typs `std::optional<int>` kann beispielsweise einen Wert des Typs `int` enthalten oder keinen Wert.

Optionals werden häufig verwendet, um entweder einen Wert darzustellen, der möglicherweise nicht vorhanden ist, oder als Rückgabetyp einer Funktion, die möglicherweise kein aussagekräftiges Ergebnis liefert.

Andere Ansätze für optional

Es gibt viele andere Ansätze zur Lösung des Problems, die von `std::optional` gelöst werden, aber keines ist vollständig: Verwenden eines Zeigers, eines Sentinels oder eines `pair<bool, T>` .

Optional gegen Zeiger

In einigen Fällen können wir einen Zeiger auf ein vorhandenes Objekt oder einen `nullptr` angeben, um einen `nullptr` anzuzeigen. Dies ist jedoch auf Fälle beschränkt, in denen Objekte bereits vorhanden sind. `optional` kann als Werttyp auch verwendet werden, um neue Objekte zurückzugeben, ohne auf die Speicherzuordnung zurückzugreifen.

Optional gegen Sentinel

Ein üblicher Ausdruck ist die Verwendung eines speziellen Werts, um anzuzeigen, dass der Wert bedeutungslos ist. Dies kann 0 oder -1 für ganzzahlige Typen sein oder `nullptr` für Zeiger. Dies reduziert jedoch den Platz für gültige Werte (Sie können nicht zwischen einer gültigen 0 und einer sinnlosen 0 unterscheiden) und viele Typen haben keine natürliche Wahl für den Sentinel-Wert.

Optional vs `std::pair<bool, T>`

Eine andere verbreitete Redewendung besteht darin, ein Paar bereitzustellen, bei dem eines der Elemente ein `bool` angibt, ob der Wert sinnvoll ist oder nicht.

Dies setzt voraus, dass der Wertetyp im Fehlerfall standardmäßig aufbaubar ist, was für einige Typen nicht möglich und für andere jedoch unerwünscht ist. Ein `optional<T>` muss im Fehlerfall nichts konstruieren.

Verwenden von Optionals, um das Fehlen eines Werts darzustellen

Vor C ++ 17 stellte das `nullptr` von Zeigern mit einem Wert von `nullptr` üblicherweise das Fehlen eines Werts dar. Dies ist eine gute Lösung für große Objekte, die dynamisch zugewiesen wurden und bereits von Zeigern verwaltet werden. Diese Lösung funktioniert jedoch nicht gut für kleine oder primitive Typen wie `int` , die selten dynamisch durch Zeiger zugewiesen oder verwaltet werden. `std::optional` bietet eine praktikable Lösung für dieses häufige Problem.

In diesem Beispiel wird `struct Person` definiert. Es ist möglich, dass eine Person ein Haustier hat, dies ist jedoch nicht erforderlich. Daher wird das `pet` Mitglied von `Person` mit einem `std::optional` Wrapper `std::optional` deklariert.

```
#include <iostream>
#include <optional>
#include <string>

struct Animal {
    std::string name;
};

struct Person {
    std::string name;
    std::optional<Animal> pet;
};

int main() {
    Person person;
    person.name = "John";

    if (person.pet) {
        std::cout << person.name << "'s pet's name is " <<
            person.pet->name << std::endl;
    }
    else {
        std::cout << person.name << " is alone." << std::endl;
    }
}
```

Verwendung von Optionals zur Darstellung des Fehlers einer Funktion

Vor C ++ 17 stellte eine Funktion normalerweise einen Fehler auf verschiedene Arten dar:

- Ein Nullzeiger wurde zurückgegeben.
    - `Delegate *App::get_delegate()` beispielsweise eine Funktion `Delegate *App::get_delegate()` auf einer `App` Instanz `Delegate *App::get_delegate()` , die keinen Delegaten hatte, würde `nullptr` .
    - Dies ist eine gute Lösung für Objekte, die dynamisch zugewiesen wurden oder groß sind und von Zeigern verwaltet werden, ist jedoch keine gute Lösung für kleine Objekte, die in der Regel stapelweise zugewiesen und durch Kopieren übergeben werden.
- Ein bestimmter Wert des Rückgabetyps wurde reserviert, um einen Fehler anzuzeigen.
    - Das Aufrufen einer Funktion `unsigned shortest_path_distance(Vertex a, Vertex b)` auf zwei nicht verbundenen Scheitelpunkten kann den Wert Null zurückgeben, um diese Tatsache anzuzeigen.
- Der Wert wurde mit einem `bool` gepaart, um anzuzeigen, dass der zurückgegebene Wert

sinnvoll ist.

- Ein Aufruf einer Funktion `std::pair<int, bool> parse(const std::string &str)` mit einem String-Argument, das keine Ganzzahl ist, würde ein Paar mit einem undefined `int` und einem `bool` auf `false` .

In diesem Beispiel erhält John zwei Haustiere, Fluffy und Furball. Die Funktion `Person::pet_with_name()` wird dann aufgerufen, um Johns Whiskers abzurufen. Da John kein Haustier mit dem Namen Whiskers hat, schlägt die Funktion fehl und `std::nullopt` wird `std::nullopt` zurückgegeben.

```
#include <iostream>
#include <optional>
#include <string>
#include <vector>

struct Animal {
    std::string name;
};

struct Person {
    std::string name;
    std::vector<Animal> pets;

    std::optional<Animal> pet_with_name(const std::string &name) {
        for (const Animal &pet : pets) {
            if (pet.name == name) {
                return pet;
            }
        }
        return std::nullopt;
    }
};

int main() {
    Person john;
    john.name = "John";

    Animal fluffy;
    fluffy.name = "Fluffy";
    john.pets.push_back(fluffy);

    Animal furball;
    furball.name = "Furball";
    john.pets.push_back(furball);

    std::optional<Animal> whiskers = john.pet_with_name("Whiskers");
    if (whiskers) {
        std::cout << "John has a pet named Whiskers." << std::endl;
    }
    else {
        std::cout << "Whiskers must not belong to John." << std::endl;
    }
}
```

optional als Rückgabewert

```
std::optional<float> divide(float a, float b) {
  if (b!=0.f) return a/b;
  return {};
}
```

Hier geben wir entweder den Bruch `a/b` , aber wenn es nicht definiert ist (wäre unendlich), geben wir stattdessen das leere optional zurück.

Ein komplexerer Fall:

```
template<class Range, class Pred>
auto find_if( Range&& r, Pred&& p ) {
  using std::begin; using std::end;
  auto b = begin(r), e = end(r);
  auto r = std::find_if(b, e , p );
  using iterator = decltype(r);
  if (r==e)
    return std::optional<iterator>();
  return std::optional<iterator>(r);
}
template<class Range, class T>
auto find( Range&& r, T const& t ) {
  return find_if( std::forward<Range>(r), [&t](auto&& x){return x==t;} );
}
```

`find( some_range, 7 )` durchsucht den Container oder den Bereich `some_range` nach etwas, das der Nummer `7` . `find_if` tut es mit einem Prädikat.

Es wird entweder ein leeres optionales Element zurückgegeben, wenn es nicht gefunden wurde, oder ein optionales Element, das einen Iterator für das Element enthält, falls es gefunden wurde.

Dies ermöglicht Ihnen Folgendes:

```
if (find( vec, 7 )) {
  // code
}
```

oder auch

```
if (auto oit = find( vec, 7 )) {
  vec.erase(*oit);
}
```

ohne mit Anfang / Ende Iteratoren und Tests herumspielen zu müssen.

value_or

```
void print_name( std::ostream& os, std::optional<std::string> const& name ) {
  std::cout "Name is: " << name.value_or("<name missing>") << '\n';
}
```

`value_or` entweder den im optionalen Wert gespeicherten Wert zurück oder das Argument, wenn

dort nichts gespeichert ist.

Auf diese Weise können Sie die Option vielleicht-null verwenden und ein Standardverhalten angeben, wenn Sie tatsächlich einen Wert benötigen. Auf diese Weise kann die Entscheidung "Standardverhalten" an den Punkt zurückgedrängt werden, an dem sie am besten getroffen wird und sofort benötigt wird, anstatt einen Standardwert zu erzeugen, der tief im Innern einer Engine liegt.

std :: optional online lesen: https://riptutorial.com/de/cplusplus/topic/2423/std----optional

Kapitel 120: std :: pair

Examples

Ein Paar erstellen und auf die Elemente zugreifen

Paar ermöglicht es uns, zwei Objekte als ein Objekt zu behandeln. Mit Hilfe der Vorlagenfunktion `std::make_pair` können Paare leicht `std::make_pair` .

Alternativ können Sie ein Paar erstellen und dessen Elemente (`first` und `second` Element) später zuweisen.

```
#include <iostream>
#include <utility>

int main()
{
    std::pair<int,int> p = std::make_pair(1,2); //Creating the pair
    std::cout << p.first << " " << p.second << std::endl; //Accessing the elements




    //We can also create a pair and assign the elements later
    std::pair<int,int> p1;
    p1.first = 3;
    p1.second = 4;
    std::cout << p1.first << " " << p1.second << std::endl;

    //We can also create a pair using a constructor
    std::pair<int,int> p2 = std::pair<int,int>(5, 6);
    std::cout << p2.first << " " << p2.second << std::endl;

    return 0;
}
```

Operatoren vergleichen

Parameter dieser Operatoren sind `lhs` und `rhs`

- `operator==` prüft, ob beide Elemente des `lhs` und `rhs` Paares gleich sind. Der Rückgabewert ist `true` wenn sowohl `lhs.first == rhs.first lhs.second == rhs.second` , andernfalls `false`

```
std::pair<int, int> p1 = std::make_pair(1, 2);
std::pair<int, int> p2 = std::make_pair(2, 2);

if (p1 == p2)
    std::cout << "equals";
else
    std::cout << "not equal"//statement will show this, because they are not identical
```

- `operator!=` prüft, ob Elemente auf dem Paar `lhs` und `rhs` nicht gleich sind. Der Rückgabewert

ist `true` wenn entweder `lhs.first != rhs.first` ODER `lhs.second != rhs.second` , andernfalls `false` .

- `operator<` - Tests , wenn `lhs.first<rhs.first` kehrt `true` . Andernfalls, wenn `rhs.first<lhs.first false` zurückgibt. Andernfalls, wenn `lhs.second<rhs.second true` zurückgibt, andernfalls `false` .
- `operator<=` zurück `!(rhs<lhs)`
- `operator>` gibt `rhs<lhs`
- `operator>=` zurück `!(lhs<rhs)`

Ein anderes Beispiel mit Containern von Paaren. Es verwendet den `operator<` da der Container sortiert werden muss.

```
#include <iostream>
#include <utility>
#include <vector>
#include <algorithm>
#include <string>

int main()
{
    std::vector<std::pair<int, std::string>> v = { {2, "baz"},
                                                   {2, "bar"},
                                                   {1, "foo"} };
    std::sort(v.begin(), v.end());

    for(const auto& p: v) {
        std::cout << "(" << p.first << "," << p.second << ") ";
        //output: (1,foo) (2,bar) (2,baz)
    }
}
```

std :: pair online lesen: https://riptutorial.com/de/cplusplus/topic/4834/std----pair

Kapitel 121: std :: set und std :: multiset

Einführung

`set` ist ein Typ von Container, dessen Elemente sortiert und eindeutig sind. `multiset` ist ähnlich, aber bei `multiset` können mehrere Elemente denselben Wert haben.

Bemerkungen

In diesen Beispielen wurden verschiedene C ++ - Stile verwendet. Seien Sie vorsichtig, wenn Sie einen C ++ 98-Compiler verwenden. Ein Teil dieses Codes ist möglicherweise nicht verwendbar.

Examples

Werte in einen Satz einfügen

Drei verschiedene Einfügemethoden können mit Sets verwendet werden.

- Zuerst eine einfache Einfügung des Wertes. Diese Methode gibt ein Paar zurück, mit dem der Aufrufer prüfen kann, ob die Einfügung tatsächlich erfolgt ist.
- Zweitens eine Einfügung, indem ein Hinweis gegeben wird, wo der Wert eingefügt wird. Das Ziel besteht darin, die Einfügungszeit in einem solchen Fall zu optimieren, aber zu wissen, wo ein Wert eingefügt werden sollte, ist nicht üblich. **Seien Sie in diesem Fall vorsichtig. Die Art, einen Hinweis zu geben, unterscheidet sich von Compiler-Versionen** .
- Schließlich können Sie einen Wertebereich einfügen, indem Sie einen Start- und einen Endzeiger angeben. Die Startnummer wird in die Einfügung aufgenommen, die Endung wird ausgeschlossen.

```
#include <iostream>
#include <set>

int main ()
{
  std::set<int> sut;
  std::set<int>::iterator it;
  std::pair<std::set<int>::iterator,bool> ret;

  // Basic insert
  sut.insert(7);
  sut.insert(5);
  sut.insert(12);

  ret = sut.insert(23);
  if (ret.second==true)
    std::cout << "# 23 has been inserted!" << std::endl;

  ret = sut.insert(23); // since it's a set and 23 is already present in it, this insert
should fail
  if (ret.second==false)
```

```
    std::cout << "# 23 already present in set!" << std::endl;

  // Insert with hint for optimization
  it = sut.end();
  // This case is optimized for C++11 and above
  // For earlier version, point to the element preceding your insertion
  sut.insert(it, 30);

  // inserting a range of values
  std::set<int> sut2;
  sut2.insert(20);
  sut2.insert(30);
  sut2.insert(45);
  std::set<int>::iterator itStart = sut2.begin();
  std::set<int>::iterator itEnd = sut2.end();

  sut.insert (itStart, itEnd); // second iterator is excluded from insertion

  std::cout << std::endl << "Set under test contains:" << std::endl;
  for (it = sut.begin(); it != sut.end(); ++it)
  {
    std::cout << *it << std::endl;
  }

  return 0;
}
```

Ausgabe wird sein:

```
# 23 has been inserted!

# 23 already present in set!



Set under test contains:

5

7

12

20

23

30

45
```

Werte in ein Multiset einfügen

Alle Einfügemethoden aus Sets gelten auch für Multisets. Es besteht jedoch eine weitere Möglichkeit, die eine initializer_list bereitstellt:

```
auto il = { 7, 5, 12 };
```

```
std::multiset<int> msut;
msut.insert(il);
```

Ändern der Standardart einer Gruppe

`set` und `multiset` verfügen über Standardvergleichsmethoden, in einigen Fällen müssen Sie sie jedoch überladen.

Stellen wir uns vor, wir speichern String-Werte in einer Menge, aber wir wissen, dass diese Strings nur numerische Werte enthalten. Standardmäßig handelt es sich bei der Sortierung um einen lexikographischen Stringvergleich, sodass die Sortierung nicht mit der numerischen Sortierung übereinstimmt. Wenn Sie eine Sortierung verwenden möchten, die dem entspricht, was Sie mit `int` Werten hätten, benötigen Sie einen Functor, um die Compare-Methode zu überladen:

```
#include <iostream>
#include <set>
#include <stdlib.h>

struct custom_compare final
{
    bool operator() (const std::string& left, const std::string& right) const
    {
        int nLeft = atoi(left.c_str());
        int nRight = atoi(right.c_str());
        return nLeft < nRight;
    }
};

int main ()
{
    std::set<std::string> sut({"1", "2", "5", "23", "6", "290"});

    std::cout << "### Default sort on std::set<std::string> :" << std::endl;
    for (auto &&data: sut)
        std::cout << data << std::endl;

    std::set<std::string, custom_compare> sut_custom({"1", "2", "5", "23", "6", "290"},
                                                     custom_compare{}); //< Compare object
optional as its default constructible.

    std::cout << std::endl << "### Custom sort on set :" << std::endl;
    for (auto &&data : sut_custom)
        std::cout << data << std::endl;

    auto compare_via_lambda = [](auto &&lhs, auto &&rhs){ return lhs > rhs; };
    using set_via_lambda = std::set<std::string, decltype(compare_via_lambda)>;
    set_via_lambda sut_reverse_via_lambda({"1", "2", "5", "23", "6", "290"},
                                          compare_via_lambda);

    std::cout << std::endl << "### Lambda sort on set :" << std::endl;
    for (auto &&data : sut_reverse_via_lambda)
        std::cout << data << std::endl;

    return 0;
}
```

Ausgabe wird sein:

```
### Default sort on std::set<std::string> :
1
2
23
290
5
6
### Custom sort on set :
1
2
5
6
23
290

### Lambda sort on set :
6
5
290
23
2
1
```

Im obigen Beispiel gibt es drei verschiedene Möglichkeiten, Vergleichsoperationen zum `std::set` hinzuzufügen. Jede davon ist in ihrem eigenen Kontext nützlich.

Standardsortierung

Der Vergleichsoperator des Schlüssels wird verwendet (erstes Vorlagenargument). Häufig bietet der Schlüssel bereits einen guten Standard für die Funktion `std::less<T>` . Wenn diese Funktion nicht spezialisiert ist, verwendet sie den `operator<` des Objekts. Dies ist besonders nützlich, wenn auch anderer Code versucht, eine bestimmte Reihenfolge zu verwenden, da dies die Konsistenz über die gesamte Codebasis ermöglicht.

Wenn Sie den Code auf diese Weise schreiben, verringert sich der Aufwand für die Aktualisierung Ihres Codes, wenn der Schlüssel die API ändert, z. B. eine Klasse mit 2 Mitgliedern, die in eine Klasse mit 3 Mitgliedern geändert wird. Durch das Aktualisieren des `operator<` in der Klasse werden alle Vorkommen aktualisiert.

Wie zu erwarten, ist die Verwendung der Standardsortierung eine sinnvolle Standardeinstellung.

Benutzerdefinierte Sortierung

Das Hinzufügen einer benutzerdefinierten Sortierung über ein Objekt mit einem Vergleichsoperator wird häufig verwendet, wenn der Standardvergleich nicht entspricht. Im obigen Beispiel liegt dies daran, dass sich die Zeichenfolgen auf Ganzzahlen beziehen. In anderen Fällen wird es häufig verwendet, wenn Sie (intelligente) Zeiger basierend auf dem Objekt, auf das sie sich beziehen, vergleichen möchten oder weil Sie zum Vergleichen unterschiedliche Einschränkungen benötigen (Beispiel: Vergleich von `std::pair` mit dem Wert von `first`).

Beim Erstellen eines Vergleichsoperators sollte dies eine stabile Sortierung sein. Wenn sich das Ergebnis des Vergleichsoperators nach dem Einfügen ändert, haben Sie undefiniertes Verhalten. In der Regel sollte Ihr Vergleichsoperator nur die konstanten Daten (const-Member, const-Funktionen ...) verwenden.

Wie im obigen Beispiel werden Sie Klassen ohne Mitglieder häufig als Vergleichsoperatoren begegnen. Dies führt zu Standardkonstruktoren und Kopierkonstruktoren. Mit dem Standardkonstruktor können Sie die Instanz zur Konstruktionszeit weglassen, und der Kopierkonstruktor ist erforderlich, da der Satz eine Kopie des Vergleichsoperators übernimmt.

Lambda Art

Lambdas sind eine kürzere Möglichkeit, Funktionsobjekte zu schreiben. Dies ermöglicht das Schreiben des Vergleichsoperators in weniger Zeilen, wodurch der gesamte Code lesbarer wird.

Der Nachteil der Verwendung von Lambdas besteht darin, dass jedes Lambda zur Kompilierzeit einen bestimmten Typ erhält, sodass der `decltype(lambda)` bei jeder Kompilierung derselben Kompilierungseinheit (cpp-Datei) anders ist als bei mehreren Kompilierungseinheiten (wenn sie über eine Headerdatei eingefügt werden)). Aus diesem Grund wird empfohlen, Funktionsobjekte als Vergleichsoperator zu verwenden, wenn sie in Header-Dateien verwendet werden.

Diese Konstruktion wird häufig angetroffen, wenn stattdessen ein `std::set` im lokalen Gültigkeitsbereich einer Funktion verwendet wird, während das Funktionsobjekt bevorzugt wird, wenn es als Funktionsargument oder Klassenmitglied verwendet wird.

Andere Sortieroptionen

Da der Vergleichsoperator von `std::set` ein Vorlagenargument ist, können alle aufrufbaren Objekte als Vergleichsoperator verwendet werden, und die obigen Beispiele sind nur Sonderfälle. Die einzigen Einschränkungen, die diese aufrufbaren Objekte haben, sind:

- Sie müssen kopierfähig sein
- Sie müssen mit 2 Argumenten des Schlüsseltyps aufrufbar sein. (Implizite Konvertierungen sind zulässig, werden jedoch nicht empfohlen, da dies die Leistung beeinträchtigen kann.)

Suche nach Werten in Set und Multiset

Es gibt mehrere Möglichkeiten, einen bestimmten Wert in `std::set` oder in `std::multiset` zu suchen:

Um den Iterator des ersten Vorkommens eines Schlüssels zu erhalten, kann die Funktion `find()` verwendet werden. Es gibt `end()` wenn der Schlüssel nicht vorhanden ist.

```
std::set<int> sut;
sut.insert(10);
sut.insert(15);
sut.insert(22);
sut.insert(3); // contains 3, 10, 15, 22
```

```
  auto itS = sut.find(10); // the value is found, so *itS == 10
  itS = sut.find(555); // the value is not found, so itS == sut.end()

  std::multiset<int> msut;
  sut.insert(10);
  sut.insert(15);
  sut.insert(22);
  sut.insert(15);
  sut.insert(3); // contains 3, 10, 15, 15, 22

  auto itMS = msut.find(10);
```

Ein anderer Weg wird mit der `count()` Funktion, die zählt , wie viele entsprechenden Werte in dem festgestellt worden `set` / `multiset` (im Fall eines `set` , der Rückgabewert 0 sein kann nur oder 1). Mit den gleichen Werten wie oben werden wir haben:

```
int result = sut.count(10); // result == 1
result = sut.count(555); // result == 0

result = msut.count(10); // result == 1
result = msut.count(15); // result == 2
```

Im Fall von `std::multiset` könnten mehrere Elemente den gleichen Wert haben. Um diesen Bereich zu erhalten, kann die Funktion `equal_range()` verwendet werden. Es gibt `std::pair` , das die untere Schranke des Iterators (einschließlich) und die obere Schranke (exklusiv) aufweist. Wenn der Schlüssel nicht vorhanden ist, zeigen beide Iteratoren auf den nächsten übergeordneten Wert (basierend auf der zum Sortieren des angegebenen `multiset` verwendeten Vergleichsmethode).

```
auto eqr = msut.equal_range(15);
auto st = eqr.first; // point to first element '15'
auto en = eqr.second; // point to element '22'

eqr = msut.equal_range(9); // both eqr.first and eqr.second point to element '10'
```

Werte aus einem Satz löschen

Die naheliegendste Methode, wenn Sie nur Ihr Set / Multiset auf ein leeres zurücksetzen möchten, ist `clear` :

```
  std::set<int> sut;
  sut.insert(10);
  sut.insert(15);
  sut.insert(22);
  sut.insert(3);
  sut.clear(); //size of sut is 0
```

Dann kann die `erase` verwendet werden. Es bietet einige Möglichkeiten, die der Einfügung etwas ähneln:

```
std::set<int> sut;
```

```
std::set<int>::iterator it;

sut.insert(10);
sut.insert(15);
sut.insert(22);
sut.insert(3);
sut.insert(30);
sut.insert(33);
sut.insert(45);

// Basic deletion
sut.erase(3);

// Using iterator
it = sut.find(22);
sut.erase(it);

// Deleting a range of values
it = sut.find(33);
sut.erase(it, sut.end());

std::cout << std::endl << "Set under test contains:" << std::endl;
for (it = sut.begin(); it != sut.end(); ++it)
{
  std::cout << *it << std::endl;
}
```

Ausgabe wird sein:

```
Set under test contains:

10

15

30
```

Alle diese Methoden gelten auch für `multiset` . Wenn Sie ein Element aus einem `multiset` löschen `multiset` und dieses mehrmals vorhanden ist, werden **alle entsprechenden Werte gelöscht** .

std :: set und std :: multiset online lesen: https://riptutorial.com/de/cplusplus/topic/9005/std----set-und-std----multiset

Kapitel 122: std :: string

Einführung

Strings sind Objekte, die Zeichenfolgen darstellen. Die Standard - `string` - Klasse bietet eine einfache, sichere und vielseitige Alternative zu explizitem Arrays mit `char` s , wenn sie mit Text und anderen Zeichenfolge handeln. Die C ++ - `string` Klasse ist Teil des `std` Namespaces und wurde 1998 standardisiert.

Syntax

- *// Leere String-Deklaration*

  std :: string s;

- *// Konstruieren aus const char * (C-String)*

  std :: string s ("Hallo");

  std :: string s = "Hallo";

- *// Konstruieren mit dem Copy-Konstruktor*

  std :: string s1 ("Hallo");

  std :: string s2 (s1);

- *// Konstruieren aus Teilzeichenfolge*

  std :: string s1 ("Hallo");

  std :: string s2 (s1, 0, 4); *// Kopiere 4 Zeichen von Position 0 von s1 nach s2*

- *// Konstruieren aus einem Zeichenpuffer*

  std :: string s1 ("Hallo Welt");
  std :: string s2 (s1, 5); *// Kopiere die ersten 5 Zeichen von s1 in s2*

- *// Konstruieren mit Füllkonstruktor (nur Zeichen)*

  std :: string s (5, 'a'); *// s enthält aaaaa*

- *// Konstruieren mit Range-Konstruktor und Iterator*

  std :: string s1 ("Hallo Welt");

  std :: string s2 (s1.begin (), s1.begin () + 5); *// Kopiere die ersten 5 Zeichen von s1 in s2*

Bemerkungen

Bevor Sie `std::string` , sollten Sie den Header- `string` einschließen, da er Funktionen / Operatoren / Überladungen enthält, die andere Header (z. B. `iostream`) nicht enthalten.

Die Verwendung von const char * constructor mit einem nullptr führt zu undefiniertem Verhalten.

```
std::string oops(nullptr);
std::cout << oops << "\n";
```

Die Methode `at` `std::out_of_range` Ausnahme für `std::out_of_range` wenn `index >= size()` .

Das Verhalten von `operator[]` ist etwas komplizierter, in allen Fällen hat es undefiniertes Verhalten, wenn `index > size()` , aber wenn `index == size()` :

C ++ 11

1. Bei einer nicht konstanten Zeichenfolge ist das Verhalten *undefiniert* .
2. In einer const-Zeichenfolge wird ein Verweis auf ein Zeichen mit dem Wert `CharT()` (das *Nullzeichen*) zurückgegeben.

C ++ 11

1. Ein Verweis auf ein Zeichen mit dem Wert `CharT()` (das *Nullzeichen*) wird zurückgegeben.
2. Das Ändern dieser Referenz ist *undefiniertes Verhalten* .

Seit C ++ 14 wird anstelle von `"foo"` empfohlen, `"foo"s` , da `s` ein benutzerdefiniertes literales Suffix ist , das das `const char* "foo"` in `std::string "foo"` konvertiert. .

Hinweis: Sie müssen den Namespace `std::string_literals` *oder* `std::literals` `std::string_literals` *, um das Literal* `s` *.*

Examples

Aufteilen

Verwenden Sie `std::string::substr` , um eine Zeichenfolge zu teilen. Es gibt zwei Varianten dieser Member-Funktion.

Der erste nimmt eine *Startposition ein,* von der aus der zurückgegebene Teilstring beginnen sollte. Die Startposition muss im Bereich `(0, str.length()]` gültig sein:

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(11); // "bar and world!"
```

Der zweite nimmt eine Startposition und die *Gesamtlänge* des neuen Teilstrings ein. Unabhängig

von der *Länge* wird die Teilzeichenfolge niemals über das Ende der Quellzeichenfolge hinausgehen:

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(15, 3); // "and"
```

Beachten Sie, ***dass*** Sie `substr` ohne Argumente aufrufen können. In diesem Fall wird eine genaue Kopie der Zeichenfolge zurückgegeben

```
std::string str = "Hello foo, bar and world!";
std::string newstr = str.substr(); // "Hello foo, bar and world!"
```

String ersetzen

Durch Position ersetzen

Um einen Teil eines `std::string` zu ersetzen, können Sie die Methode `replace` von `std::string` .

`replace` hat viele nützliche Überladungen:

```
//Define string
std::string str = "Hello foo, bar and world!";
std::string alternate = "Hello foobar";

//1)
str.replace(6, 3, "bar"); //"Hello bar, bar and world!"

//2)
str.replace(str.begin() + 6, str.end(), "nobody!"); //"Hello nobody!"

//3)
str.replace(19, 5, alternate, 6, 6); //"Hello foo, bar and foobar!"
```

C ++ 14

```
//4)
str.replace(19, 5, alternate, 6); //"Hello foo, bar and foobar!"
```

```
//5)
str.replace(str.begin(), str.begin() + 5, str.begin() + 6, str.begin() + 9);
//"foo foo, bar and world!"

//6)
str.replace(0, 5, 3, 'z'); //"zzz foo, bar and world!"

//7)
str.replace(str.begin() + 6, str.begin() + 9, 3, 'x'); //"Hello xxx, bar and world!"
```

C ++ 11

```
//8)
```

```
str.replace(str.begin(), str.begin() + 5, { 'x', 'y', 'z' }); //"xyz foo, bar and world!"
```

Ersetzen Sie Vorkommen einer Zeichenfolge durch eine andere Zeichenfolge

Ersetze nur das erste Vorkommen von `replace` mit `with` in `str` :

```
std::string replaceString(std::string str,
                          const std::string& replace,
                          const std::string& with){
    std::size_t pos = str.find(replace);
    if (pos != std::string::npos)
        str.replace(pos, replace.length(), with);
    return str;
}
```

Ersetzen Sie alle Vorkommen von `replace` durch `with` in `str` :

```
std::string replaceStringAll(std::string str,
                             const std::string& replace,
                             const std::string& with) {
    if(!replace.empty()) {
        std::size_t pos = 0;
        while ((pos = str.find(replace, pos)) != std::string::npos) {
            str.replace(pos, replace.length(), with);
            pos += with.length();
        }
    }
    return str;
}
```

Verkettung

Sie können `std::string` s mit den überladenen Operatoren `+` und `+=` verketten. Mit dem Operator `+` :

```
std::string hello = "Hello";
std::string world = "world";
std::string helloworld = hello + world; // "Helloworld"
```

Verwenden des Operators `+=` :

```
std::string hello = "Hello";
std::string world = "world";
hello += world; // "Helloworld"
```

Sie können auch C-Strings anhängen, einschließlich String-Literalen:

```
std::string hello = "Hello";
```

```
std::string world = "world";
const char *comma = ", ";
std::string newhelloworld = hello + comma + world + "!"; // "Hello, world!"
```

Sie können auch verwenden `push_back()` einzelne zurückdrängen `char` s:

```
std::string s = "a, b, ";
s.push_back('c'); // "a, b, c"
```

Es gibt auch `append()` , was ziemlich ähnlich ist wie `+=` :

```
std::string app = "test and ";
app.append("test"); // "test and test"
```

Zugriff auf einen Charakter

Es gibt mehrere Möglichkeiten, Zeichen aus einem `std::string` zu extrahieren, und jede ist unterschiedlich.

```
std::string str("Hello world!");
```

Operator [] (n)

Gibt eine Referenz auf das Zeichen am Index n zurück.

`std::string::operator[]` wird nicht auf Grenzen geprüft und löst keine Ausnahme aus. Der Aufrufer ist dafür verantwortlich, dass der Index im Bereich der Zeichenfolge liegt:

```
char c = str[6]; // 'w'
```

---

bei (n)

Gibt eine Referenz auf das Zeichen am Index n zurück.

`std::string::at` *wird auf* Grenzen geprüft und gibt `std::out_of_range` wenn der Index nicht im Bereich des Strings liegt:

```
char c = str.at(7); // 'o'
```

C ++ 11

Anmerkung: Beide Beispiele führen zu undefiniertem Verhalten, wenn die Zeichenfolge leer ist.

Vorderseite()

Gibt eine Referenz auf das erste Zeichen zurück:

```
char c = str.front(); // 'H'
```

---

zurück()

Gibt eine Referenz auf das letzte Zeichen zurück:

```
char c = str.back(); // '!'
```

Tokenize

Gelistet von den billigsten zur teuersten zur Laufzeit:

1. `str::strtok` ist die billigste standardisierte Tokenisierungsmethode. Sie ermöglicht auch die Änderung des Trennzeichens zwischen Tokens, verursacht jedoch 3 Probleme mit modernem C ++:

    - `std::strtok` kann nicht gleichzeitig für mehrere `strings` werden (obwohl einige Implementierungen dies unterstützen, z. `strtok_s` .: `strtok_s`)
    - Aus demselben Grund kann `std::strtok` nicht gleichzeitig auf mehreren Threads verwendet werden (dies kann jedoch durch die Implementierung definiert werden, z. B.: Die Implementierung von Visual Studio ist threadsicher).
    - Durch Aufrufen von `std::strtok` der `std::string` er arbeitet, `std::strtok` kann er nicht für `const string` s, `const char*` s oder literal strings verwendet werden, um eine dieser Zeichenketten mit `std::strtok` zu `std::strtok` oder eine `std::string` dessen Inhalt muss erhalten bleiben, die Eingabe müsste kopiert werden, dann könnte die Kopie bearbeitet werden

    Im Allgemeinen werden diese Optionen in den Zuweisungskosten der Token versteckt. Wenn jedoch der billigste Algorithmus erforderlich ist und die Schwierigkeiten von `std::strtok` nicht übermäßig sind, sollten Sie eine von Hand gesponnene Lösung in Betracht ziehen.

```
// String to tokenize
std::string str{ "The quick brown fox" };
// Vector to store tokens
vector<std::string> tokens;

for (auto i = strtok(&str[0], " "); i != NULL; i = strtok(NULL, " "))
    tokens.push_back(i);
```

Live-Beispiel

2. Der `std::istream_iterator` verwendet den Extraktionsoperator des Streams iterativ. Wenn die Eingabe `std::string` Leerzeichen getrennt ist, kann die Option `std::strtok` durch Beseitigung der Schwierigkeiten erweitert werden, wodurch die Inline-Tokenisierung ermöglicht wird, wodurch die Erzeugung eines `const vector<string>` , und indem Unterstützung für mehrere

hinzugefügt wird Leerzeichen abgrenzend:

```
// String to tokenize
const std::string str("The  quick \tbrown \nfox");
std::istringstream is(str);
// Vector to store tokens
const std::vector<std::string> tokens = std::vector<std::string>(
                                        std::istream_iterator<std::string>(is),
                                        std::istream_iterator<std::string>());
```

Live-Beispiel

3. Der `std::regex_token_iterator` verwendet einen `std::regex` um iterativ zu tokenisieren. Es bietet eine flexiblere Definition von Begrenzungszeichen. Beispiel: Nicht begrenzte Kommas und Leerzeichen:

C ++ 11

```
// String to tokenize
const std::string str{ "The ,qu\\,ick ,\tbrown, fox" };
const std::regex re{ "\\s*((?:[^\\\\,]|\\\\.)*?)\\s*(?:,|$)" };
// Vector to store tokens
const std::vector<std::string> tokens{
    std::sregex_token_iterator(str.begin(), str.end(), re, 1),
    std::sregex_token_iterator()
};
```

Live-Beispiel

Weitere `regex_token_iterator` Sie im `regex_token_iterator` Beispiel .

Konvertierung in (const) char *

Um mit `const char*` auf die Daten einer `std::string` zuzugreifen, können Sie die `c_str()` der Zeichenfolge verwenden. Beachten Sie, dass der Zeiger nur gültig ist, solange sich das Objekt `std::string` im Gültigkeitsbereich befindet und unverändert bleibt. Dies bedeutet, dass nur `const` Methoden für das Objekt aufgerufen werden können.

C ++ 17

Die `data()` Memberfunktion kann verwendet werden, um ein modifizierbares `char*` , mit dem die Daten des `std::string` Objekts bearbeitet werden können.

C ++ 11

Ein modifizierbares Zeichen `char*` kann auch erhalten werden, indem die Adresse des ersten Zeichens verwendet wird: `&s[0]` . In C ++ 11 wird garantiert, dass dies eine wohlgeformte, nullterminierte Zeichenfolge ergibt. Beachten Sie, dass `&s[0]` dann gut geformt ist, wenn `s` leer ist, wohingegen `&s.front()` undefined ist, wenn `s` leer ist.

C ++ 11

```
std::string str("This is a string.");
const char* cstr = str.c_str(); // cstr points to: "This is a string.\0"
const char* data = str.data();  // data points to: "This is a string.\0"
```

```
std::string str("This is a string.");

// Copy the contents of str to untie lifetime from the std::string object
std::unique_ptr<char []> cstr = std::make_unique<char[]>(str.size() + 1);

// Alternative to the line above (no exception safety):
// char* cstr_unsafe = new char[str.size() + 1];

std::copy(str.data(), str.data() + str.size(), cstr);
cstr[str.size()] = '\0'; // A null-terminator needs to be added

// delete[] cstr_unsafe;
std::cout << cstr.get();
```

Zeichen in einer Zeichenfolge finden

Um ein Zeichen oder eine andere Zeichenfolge zu finden, können Sie `std::string::find` . Es gibt die Position des ersten Zeichens des ersten Matches zurück. Wenn keine Übereinstimmungen gefunden wurden, gibt die Funktion `std::string::npos`

```
std::string str = "Curiosity killed the cat";
auto it = str.find("cat");

if (it != std::string::npos)
    std::cout << "Found at position: " << it << '\n';
else
    std::cout << "Not found!\n";
```

Gefunden an Position: 21

Die Suchmöglichkeiten werden um folgende Funktionen erweitert:

```
find_first_of     // Find first occurrence of characters
find_first_not_of // Find first absence of characters
find_last_of      // Find last occurrence of characters
find_last_not_of  // Find last absence of characters
```

Mit diesen Funktionen können Sie am Ende der Zeichenfolge nach Zeichen suchen und den negativen Fall (dh Zeichen, die nicht in der Zeichenfolge enthalten sind) finden. Hier ist ein Beispiel:

```
std::string str = "dog dog cat cat";
std::cout << "Found at position: " << str.find_last_of("gzx") << '\n';
```

Gefunden an Position: 6

Hinweis: Beachten Sie, dass die oben genannten Funktionen nicht nach Teilzeichenfolgen suchen, sondern nach Zeichen, die in der Suchzeichenfolge enthalten sind. In diesem Fall wurde

das letzte Vorkommen von `'g'` an Position `6` (die anderen Zeichen wurden nicht gefunden).

Zeichen am Anfang / Ende abschneiden

In diesem Beispiel sind die Header `<algorithm>` , `<locale>` und `<utility>` erforderlich.

C ++ 11

Um eine Sequenz oder String *trim* bedeutet alle vorderen und hinteren Elemente (oder Zeichen) zu entfernen , um ein bestimmtes Prädikat übereinstimmt. Wir schneiden zuerst die nachlaufenden Elemente ab, da keine Elemente bewegt werden müssen, und dann die führenden Elemente. Beachten Sie, dass die folgenden Verallgemeinerungen für alle Arten von `std::basic_string` (z. B. `std::string` und `std::wstring`) und aus Versehen auch für `std::basic_string` (z. B. `std::vector` und `std::list`) `std::wstring` .

```
template <typename Sequence, // any basic_string, vector, list etc.
          typename Pred>     // a predicate on the element (character) type
Sequence& trim(Sequence& seq, Pred pred) {
    return trim_start(trim_end(seq, pred), pred);
}
```

Beim Trimmen der nachfolgenden Elemente wird das *letzte* Element gefunden, das nicht mit dem Vergleichselement übereinstimmt, und von dort aus gelöscht:

```
template <typename Sequence, typename Pred>
Sequence& trim_end(Sequence& seq, Pred pred) {
    auto last = std::find_if_not(seq.rbegin(),
                                 seq.rend(),
                                 pred);
    seq.erase(last.base(), seq.end());
    return seq;
}
```

Beim Trimmen der führenden Elemente wird das *erste* Element gefunden, das nicht mit dem Prädikat übereinstimmt, und bis dahin gelöscht:

```
template <typename Sequence, typename Pred>
Sequence& trim_start(Sequence& seq, Pred pred) {
    auto first = std::find_if_not(seq.begin(),
                                  seq.end(),
                                  pred);
    seq.erase(seq.begin(), first);
    return seq;
}
```

Um das Obige auf das Trimmen von Whitespace in einem `std::string` , können wir die Funktion `std::isspace()` als Prädikat verwenden:

```
std::string& trim(std::string& str, const std::locale& loc = std::locale()) {
    return trim(str, [&loc](const char c){ return std::isspace(c, loc); });
}
```

```
std::string& trim_start(std::string& str, const std::locale& loc = std::locale()) {
    return trim_start(str, [&loc](const char c){ return std::isspace(c, loc); });
}

std::string& trim_end(std::string& str, const std::locale& loc = std::locale()) {
    return trim_end(str, [&loc](const char c){ return std::isspace(c, loc); });
}
```

In ähnlicher Weise können wir die Funktion `std::iswspace()` für `std::wstring` usw. verwenden.

Wenn Sie eine *neue* Sequenz erstellen möchten, die eine zugeschnittene Kopie ist, können Sie eine separate Funktion verwenden:

```
template <typename Sequence, typename Pred>
Sequence trim_copy(Sequence seq, Pred pred) { // NOTE: passing seq by value
    trim(seq, pred);
    return seq;
}
```

Lexikographischer Vergleich

Zwei `std::string` s können lexikographisch mit den Operatoren `==` `!=` , `<` , `<=` , `>` Und `>=` verglichen werden:

```
std::string str1 = "Foo";
std::string str2 = "Bar";

assert(!(str1 < str2));
assert(str > str2);
assert(!(str1 <= str2));
assert(str1 >= str2);
assert(!(str1 == str2));
assert(str1 != str2);
```

Alle diese Funktionen verwenden die zugrunde liegende Methode `std::string::compare()` , um den Vergleich durchzuführen und zur Vereinfachung boolesche Werte zurückzugeben. Die Funktionsweise dieser Funktionen kann unabhängig von der tatsächlichen Implementierung wie folgt interpretiert werden:

- Operator `==` :

  Wenn `str1.length() == str2.length()` und jedes Zeichenpaar übereinstimmt, wird `true` , andernfalls `false` .

- Operator `!=` :

  Wenn `str1.length() != str2.length()` oder ein Zeichenpaar nicht übereinstimmt, wird `true` , andernfalls `false` .

- Operator `<` oder Operator `>` :

  Findet das erste unterschiedliche Zeichenpaar, vergleicht sie und gibt das boolesche

Ergebnis zurück.

- Operator `<=` oder Operator `>=` :

  Findet das erste unterschiedliche Zeichenpaar, vergleicht sie und gibt das boolesche Ergebnis zurück.

Hinweis: Unter dem Begriff **Zeichenpaar** sind die entsprechenden Zeichen in beiden Zeichenfolgen derselben Position zu verstehen. Zum besseren Verständnis `str1` , wenn zwei Beispielketten `str1` und `str2` sind und ihre Längen jeweils `n` und `m` sind, die Zeichenpaare beider Strings jeweils `str1[i]` und `str2[i]` Paare, wobei *i = 0, 1, 2* `str2[i]` *.., max (n, m)* . Wenn für ein *i,* wenn das entsprechende Zeichen nicht existiert, das heißt, wenn *i* größer als oder gleich `n` oder `m` , würde es als der niedrigste Wert betrachtet werden.

Hier ist ein Beispiel für die Verwendung von `<` :

```
std::string str1 = "Barr";
std::string str2 = "Bar";

assert(str2 < str1);
```

Die Schritte sind wie folgt:

1. Vergleichen Sie die ersten Zeichen, `'B' == 'B'` - fahren Sie fort.
2. Vergleichen Sie die zweiten Zeichen, `'a' == 'a'` - weiter.
3. Vergleichen Sie die dritten Zeichen, `'r' == 'r'` - fahren Sie fort.
4. Der `str2` Bereich ist jetzt erschöpft, während der `str1` Bereich noch Zeichen enthält. Somit ist `str2 < str1` .

Konvertierung in std :: wstring

In C ++ werden Zeichenfolgen dargestellt, indem die Klasse `std::basic_string` auf einen nativen Zeichentyp spezialisiert wird. Die zwei Hauptsammlungen, die von der Standardbibliothek definiert werden, sind `std::string` und `std::wstring` :

- `std::string` wird mit Elementen des Typs `char`
- `std::wstring` besteht aus Elementen des Typs `wchar_t`

Um zwischen den beiden Typen zu konvertieren, verwenden Sie `wstring_convert` :

```
#include <string>
#include <codecvt>
#include <locale>

std::string input_str = "this is a -string-, which is a sequence based on the -char- type.";
std::wstring input_wstr = L"this is a -wide- string, which is based on the -wchar_t- type.";

// conversion
std::wstring str_turned_to_wstr =
```

```
std::wstring_convert<std::codecvt_utf8<wchar_t>>().from_bytes(input_str);

std::string wstr_turned_to_str =
std::wstring_convert<std::codecvt_utf8<wchar_t>>().to_bytes(input_wstr);
```

Um die Benutzerfreundlichkeit und / oder Lesbarkeit zu verbessern, können Sie Funktionen zur Durchführung der Konvertierung definieren:

```
#include <string>
#include <codecvt>
#include <locale>

using convert_t = std::codecvt_utf8<wchar_t>;
std::wstring_convert<convert_t, wchar_t> strconverter;

std::string to_string(std::wstring wstr)
{
    return strconverter.to_bytes(wstr);
}

std::wstring to_wstring(std::string str)
{
    return strconverter.from_bytes(str);
}
```

Verwendungsbeispiel:

```
std::wstring a_wide_string = to_wstring("Hello World!");
```

Das ist sicherlich lesbarer als
`std::wstring_convert<std::codecvt_utf8<wchar_t>>().from_bytes("Hello World!")` .

Bitte beachten Sie, dass `char` und `wchar_t` keine Kodierung implizieren und keine Angabe der Größe in Byte enthalten. Beispielsweise ist `wchar_t` im Allgemeinen als 2-Byte-Datentyp implementiert und enthält in der Regel UTF-16-codierte Daten unter Windows (oder UCS-2 in Versionen vor Windows 2000) und als 4-Byte-Datentyp, der mit UTF-32 unter codiert ist Linux. Dies steht im Gegensatz zu den neueren Typen `char16_t` und `char32_t` , die in C ++ 11 eingeführt wurden und die garantiert groß genug sind, um UTF16- oder UTF32-Zeichen (oder genauer *Codepunkt*) aufzunehmen.

Verwenden der Klasse std :: string_view

C ++ 17

In C ++ 17 wird `std::string_view` , bei dem es sich einfach um einen nicht besitzenden Bereich von `const char` `std::string_view` , der entweder als Zeigerpaar oder als Zeiger und als Länge implementiert werden kann. Dies ist ein überlegener Parametertyp für Funktionen, für die nicht modifizierbare Zeichenfolgendaten erforderlich sind. Vor C ++ 17 gab es dafür drei Möglichkeiten:

```
void foo(std::string const& s);      // pre-C++17, single argument, could incur
                                     // allocation if caller's data was not in a string
```

```
                                    // (e.g. string literal or vector<char> )

void foo(const char* s, size_t len); // pre-C++17, two arguments, have to pass them
                                    // both everywhere

void foo(const char* s);            // pre-C++17, single argument, but need to call
                                    // strlen()

template <class StringT>
void foo(StringT const& s);         // pre-C++17, caller can pass arbitrary char data
                                    // provider, but now foo() has to live in a header
```

Alle diese können ersetzt werden durch:

```
void foo(std::string_view s);       // post-C++17, single argument, tighter coupling
                                    // zero copies regardless of how caller is storing
                                    // the data
```

Beachten Sie, dass `std::string_view` ***die*** *zugrunde liegenden Daten* ***nicht*** *ändern kann* .

`string_view` ist nützlich, wenn Sie unnötige Kopien vermeiden möchten.

Es bietet eine nützliche Untermenge der Funktionalität, die von `std::string` wird, obwohl einige Funktionen sich unterschiedlich verhalten:

```
std::string str = "lllloooonnnnggggg sssstttrrriiinnnggg"; //A really long string

//Bad way - 'string::substr' returns a new string (expensive if the string is long)
std::cout << str.substr(15, 10) << '\n';

//Good way - No copies are created!
std::string_view view = str;

// string_view::substr returns a new string_view
std::cout << view.substr(15, 10) << '\n';
```

Durchlaufen jedes Charakters

C ++ 11

`std::string` unterstützt Iteratoren. Daher können Sie eine *range-basierte* Schleife verwenden, um jedes Zeichen zu durchlaufen:

```
std::string str = "Hello World!";
for (auto c : str)
    std::cout << c;
```

Sie können eine "traditionelle" `for` Schleife verwenden, um jedes Zeichen zu durchlaufen:

```
std::string str = "Hello World!";
for (std::size_t i = 0; i < str.length(); ++i)
    std::cout << str[i];
```

Konvertierung in Ganzzahlen / Gleitkommatypen

Ein `std::string` , der eine Zahl enthält, kann mithilfe von Konvertierungsfunktionen in einen Integer-Typ oder einen Gleitkommatyp konvertiert werden.

Beachten Sie, dass alle diese Funktionen die Analyse der Eingabezeichenfolge beenden, sobald sie auf ein nicht numerisches Zeichen stoßen, so dass `"123abc"` in `123` .

Die Funktionsfamilie `std::ato*` konvertiert Zeichenketten im C-Stil (Zeichenarrays) in Ganzzahl- oder Fließkommatypen:

```
std::string ten = "10";

double num1 = std::atof(ten.c_str());
int num2 = std::atoi(ten.c_str());
long num3 = std::atol(ten.c_str());
```

C ++ 11

```
long long num4 = std::atoll(ten.c_str());
```

Die Verwendung dieser Funktionen wird jedoch nicht empfohlen, da sie `0` wenn die Zeichenfolge nicht analysiert werden kann. Dies ist schlecht, da `0` auch ein gültiges Ergebnis sein kann, wenn die Eingabezeichenfolge beispielsweise "0" war. Daher kann nicht festgestellt werden, ob die Konvertierung tatsächlich fehlgeschlagen ist.

Die neuere Funktionsfamilie `std::sto*` konvertiert `std::string` s in Ganzzahl- oder Fließkommatypen und löst Ausnahmen aus, wenn sie ihre Eingabe nicht analysieren konnten. *Sie sollten diese Funktionen nach Möglichkeit verwenden* :

C ++ 11

```
std::string ten = "10";

int num1 = std::stoi(ten);
long num2 = std::stol(ten);
long long num3 = std::stoll(ten);

float num4 = std::stof(ten);
double num5 = std::stod(ten);
long double num6 = std::stold(ten);
```

Darüber hinaus behandeln diese Funktionen im Gegensatz zur `std::ato*` -Familie auch Oktal- und Hex-Strings. Der zweite Parameter ist ein Zeiger auf das erste nicht konvertierte Zeichen in der Eingabezeichenfolge (hier nicht dargestellt), und der dritte Parameter ist die zu verwendende Basis. `0` ist die automatische Erkennung von Oktal (beginnend mit `0`) und Hex (beginnend mit `0x` oder `0X`), und jeder andere Wert ist die zu verwendende Basis

```
std::string ten = "10";
std::string ten_octal = "12";
```

```
std::string ten_hex = "0xA";

int num1 = std::stoi(ten, 0, 2); // Returns 2
int num2 = std::stoi(ten_octal, 0, 8); // Returns 10
long num3 = std::stol(ten_hex, 0, 16);  // Returns 10
long num4 = std::stol(ten_hex);  // Returns 0
long num5 = std::stol(ten_hex, 0, 0); // Returns 10 as it detects the leading 0x
```

Konvertierung zwischen Zeichenkodierungen

Das Konvertieren zwischen Codierungen ist mit C ++ 11 einfach und die meisten Compiler können plattformübergreifend mit den `<codecvt>` und `<locale> <codecvt>` .

```
#include <iostream>
#include <codecvt>
#include <locale>
#include <string>
using namespace std;

int main() {
    // converts between wstring and utf8 string
    wstring_convert<codecvt_utf8_utf16<wchar_t>> wchar_to_utf8;
    // converts between u16string and utf8 string
    wstring_convert<codecvt_utf8_utf16<char16_t>, char16_t> utf16_to_utf8;

    wstring wstr = L"foobar";
    string utf8str = wchar_to_utf8.to_bytes(wstr);
    wstring wstr2 = wchar_to_utf8.from_bytes(utf8str);

    wcout << wstr << endl;
    cout << utf8str << endl;
    wcout << wstr2 << endl;

    u16string u16str = u"foobar";
    string utf8str2 = utf16_to_utf8.to_bytes(u16str);
    u16string u16str2 = utf16_to_utf8.from_bytes(utf8str2);

    return 0;
}
```

Beachten Sie, dass Visual Studio 2015 Unterstützung für diese Konvertierung bietet. Ein Fehler in der Bibliotheksimplementierung erfordert die Verwendung einer anderen Vorlage für `wstring_convert` wenn mit `char16_t` :

```
using utf16_char = unsigned short;
wstring_convert<codecvt_utf8_utf16<utf16_char>, utf16_char> conv_utf8_utf16;

void strings::utf16_to_utf8(const std::u16string& utf16, std::string& utf8)
{
  std::basic_string<utf16_char> tmp;
  tmp.resize(utf16.length());
  std::copy(utf16.begin(), utf16.end(), tmp.begin());
  utf8 = conv_utf8_utf16.to_bytes(tmp);
}
void strings::utf8_to_utf16(const std::string& utf8, std::u16string& utf16)
{
  std::basic_string<utf16_char> tmp = conv_utf8_utf16.from_bytes(utf8);
```

```
  utf16.clear();
  utf16.resize(tmp.length());
  std::copy(tmp.begin(), tmp.end(), utf16.begin());
}
```

Prüfen, ob eine Zeichenfolge ein Präfix einer anderen ist

C ++ 14

In C ++ 14 kann dies problemlos mit `std::mismatch` wobei das erste Paar aus zwei Bereichen das nicht übereinstimmende Paar zurückgibt:

```
std::string prefix = "foo";
std::string string = "foobar";

bool isPrefix = std::mismatch(prefix.begin(), prefix.end(),
    string.begin(), string.end()).first == prefix.end();
```

Beachten Sie, dass vor Andern von C ++ 14 eine anderthalbminütige Version von `mismatch()` vorhanden war. Dies ist jedoch nicht sicher, wenn die zweite Zeichenfolge die kürzere der beiden ist.

C ++ 14

Wir können immer noch die Range-and-a-half-Version von `std::mismatch()` , aber wir müssen zuerst prüfen, ob der erste String höchstens so groß ist wie der zweite:

```
bool isPrefix = prefix.size() <= string.size() &&
    std::mismatch(prefix.begin(), prefix.end(),
        string.begin(), string.end()).first == prefix.end();
```

C ++ 17

Mit `std::string_view` können wir den gewünschten direkten Vergleich schreiben, ohne sich um den Zuordnungsaufwand oder das `std::string_view` Kopien kümmern zu müssen:

```
bool isPrefix(std::string_view prefix, std::string_view full)
{
    return prefix == full.substr(0, prefix.size());
}
```

Konvertierung in std :: string

`std::ostringstream` kann jeder Streamable-Typ in eine String-Darstellung konvertiert werden, indem das Objekt in ein `std::ostringstream` Objekt `std::ostringstream` wird (mit dem Stream-Einfügungsoperator `<<`) und anschließend der gesamte `std::ostringstream` in einen `std::string` konvertiert wird `std::string` .

Für `int` zum Beispiel:

```
#include <sstream>

int main()
{
    int val = 4;
    std::ostringstream str;
    str << val;
    std::string converted = str.str();
    return 0;
}
```

Schreiben Sie Ihre eigene Konvertierungsfunktion:

```
template<class T>
std::string toString(const T& x)
{
  std::ostringstream ss;
  ss << x;
  return ss.str();
}
```

funktioniert, ist jedoch nicht für leistungskritischen Code geeignet.

Benutzerdefinierte Klassen können den Stream-Einfügeoperator implementieren, falls dies gewünscht ist:

```
std::ostream operator<<( std::ostream& out, const A& a )
{
    // write a string representation of a to out
    return out;
}
```

C ++ 11

Abgesehen von Streams können Sie seit C ++ 11 auch die Funktion `std::to_string` (und `std::to_wstring`) verwenden, die für alle grundlegenden Typen überladen ist und die String-Darstellung ihres Parameters zurückgibt.

```
std::string s = to_string(0x12f3);  // after this the string s contains "4851"
```

std :: string online lesen: https://riptutorial.com/de/cplusplus/topic/488/std----string

Kapitel 123: std :: variant

Bemerkungen

Variante ist ein Ersatz für rohe `union` Einsatz. Es ist typensicher und weiß, um welchen Typ es sich handelt, und es konstruiert und zerstört die Objekte in dem Objekt sorgfältig, wenn es sollte.

Es ist fast nie leer: Nur in Eckfällen, in denen das Ersetzen des Inhalts geworfen wird und er nicht sicher zurückkehren kann, endet er in einem leeren Zustand.

Es verhält sich etwas wie ein `std::tuple` und etwas wie ein `std::optional` .

Die Verwendung von `std::get` und `std::get_if` ist normalerweise eine schlechte Idee. Die richtige Antwort ist in der Regel `std::visit` , so dass Sie direkt mit jeder Möglichkeit umgehen können. `if constexpr` innerhalb des `visit` werden kann, wenn Sie Ihr Verhalten verzweigen müssen, anstatt eine Laufzeitprüfung `if constexpr` , wird überprüft, ob der `visit` effizienter ist.

Examples

Grundsätzliche Verwendung von std :: variant

Dadurch wird eine Variante (eine markierte Union) erstellt, in der entweder ein `int` oder ein `string` gespeichert werden kann.

```
std::variant< int, std::string > var;
```

Wir können einen von beiden Typen darin speichern:

```
var = "hello"s;
```

Und wir können auf die Inhalte über `std::visit` zugreifen:

```
// Prints "hello\n":
visit( [](auto&& e) {
  std::cout << e << '\n';
}, var );
```

durch Einleiten eines polymorphen Lambda oder eines ähnlichen Funktionsobjekts.

Wenn wir sicher sind, um welchen Typ es sich handelt, können wir ihn bekommen:

```
auto str = std::get<std::string>(var);
```

aber das wird werfen, wenn wir es falsch verstehen. `get_if` :

```
auto* str  = std::get_if<std::string>(&var);
```

`nullptr` wenn Sie falsch raten.

Varianten garantieren keine dynamische Speicherzuweisung (anders als bei den enthaltenen Typen). Dort ist nur einer der Typen gespeichert, und in seltenen Fällen (mit Ausnahmen beim Zuweisen und ohne sichere Rücknahme) kann die Variante leer werden.

Mit Varianten können Sie mehrere Wertetypen sicher und effizient in einer Variablen speichern. Sie sind im Grunde intelligente, typsichere `union` .

Erstellen Sie Pseudo-Methodenzeiger

Dies ist ein fortgeschrittenes Beispiel.

Sie können die Variante für das Löschen von Leichtgewichten verwenden.

```
template<class F>
struct pseudo_method {
  F f;
  // enable C++17 class type deduction:
  pseudo_method( F&& fin ):f(std::move(fin)) {}

  // Koenig lookup operator->*, as this is a pseudo-method it is appropriate:
  template<class Variant> // maybe add SFINAE test that LHS is actually a variant.
  friend decltype(auto) operator->*( Variant&& var, pseudo_method const& method ) {
    // var->*method returns a lambda that perfect forwards a function call,
    // behaving like a method pointer basically:
    return [&](auto&&...args)->decltype(auto) {
      // use visit to get the type of the variant:
      return std::visit(
        [&](auto&& self)->decltype(auto) {
          // decltype(x)(x) is perfect forwarding in a lambda:
          return method.f( decltype(self)(self), decltype(args)(args)... );
        },
        std::forward<Var>(var)
      );
    };
  }
};
```

Dadurch wird ein Typ erstellt, der `operator->*` mit einer `Variant` auf der linken Seite überladen.

```
// C++17 class type deduction to find template argument of `print` here.
// a pseudo-method lambda should take `self` as its first argument, then
// the rest of the arguments afterwards, and invoke the action:
pseudo_method print = [](auto&& self, auto&&...args)->decltype(auto) {
  return decltype(self)(self).print( decltype(args)(args)... );
};
```

Wenn wir nun 2 Typen mit einer `print` :

```
struct A {
  void print( std::ostream& os ) const {
    os << "A";
  }
};
```

```
struct B {
  void print( std::ostream& os ) const {
    os << "B";
  }
};
```

Beachten Sie, dass es sich nicht um Typen handelt. Wir können:

```
std::variant<A,B> var = A{};

(var->*print)(std::cout);
```

Der Anruf wird direkt an `A::print(std::cout)` für uns `A::print(std::cout)` . Wenn wir das `var` stattdessen mit `B{}` initialisieren, wird es an `B::print(std::cout)` .

Wenn wir einen neuen Typ C erstellt haben:

```
struct C {};
```

dann:

```
std::variant<A,B,C> var = A{};
(var->*print)(std::cout);
```

wird nicht kompiliert, da es keine `C.print(std::cout)` -Methode gibt.

Die oben erstreckt würde freie Funktion erlauben `print` s erkannt und verwendet werden, gegebenenfalls unter Verwendung von , `if constexpr` innerhalb des `print` pseudo-Methode.

Live-Beispiel, das momentan `boost::variant` anstelle von `std::variant` .

Konstruktion einer `std :: variant`

Dies gilt nicht für die Zuteiler.

```
struct A {};
struct B { B()=default; B(B const&)=default; B(int){}; };
struct C { C()=delete; C(int) {}; C(C const&)=default; };
struct D { D( std::initializer_list<int> ) {}; D(D const&)=default; D()=default; };

std::variant<A,B> var_ab0; // contains a A()
std::variant<A,B> var_ab1 = 7; // contains a B(7)
std::variant<A,B> var_ab2 = var_ab1; // contains a B(7)
std::variant<A,B,C> var_abc0{ std::in_place_type<C>, 7 }; // contains a C(7)
std::variant<C> var_c0; // illegal, no default ctor for C
std::variant<A,D> var_ad0( std::in_place_type<D>, {1,3,3,4} ); // contains D{1,3,3,4}
std::variant<A,D> var_ad1( std::in_place_index<0> ); // contains A{}
std::variant<A,D> var_ad2( std::in_place_index<1>, {1,3,3,4} ); // contains D{1,3,3,4}
```

std :: variant online lesen: https://riptutorial.com/de/cplusplus/topic/5239/std----variant

Kapitel 124: std :: vector

Einführung

Ein Vektor ist ein dynamisches Array mit automatisch behandeltem Speicher. Auf die Elemente in einem Vektor kann genauso effizient wie auf die Elemente in einem Array zugegriffen werden, mit dem Vorteil, dass sich die Größe von Vektoren dynamisch ändern kann.

In Bezug auf die Speicherung werden die Vektordaten (normalerweise) in einem dynamisch zugewiesenen Speicher abgelegt, wodurch ein geringerer Aufwand erforderlich ist. Umgekehrt verwenden `C-arrays` und `std::array` die automatische Speicherung relativ zum angegebenen Speicherort und haben daher keinen Overhead.

Bemerkungen

Die Verwendung eines `std::vector` erfordert die Einbeziehung des `<vector>` -Headers mit `#include <vector>` .

Elemente in einem `std::vector` werden zusammenhängend im freien Speicher gespeichert. Es sollte beachtet werden, dass, wenn Vektoren wie in `std::vector<std::vector<int> >` verschachtelt sind, die Elemente jedes Vektors zusammenhängend sind, jeder Vektor jedoch seinen eigenen zugrunde liegenden Puffer im freien Speicher zuordnet.

Examples

Initialisieren eines std :: -Vektors

Ein `std::vector` kann auf verschiedene Weise initialisiert werden, während er deklariert wird:

C ++ 11

```
std::vector<int> v{ 1, 2, 3 };  // v becomes {1, 2, 3}

// Different from std::vector<int> v(3, 6)
std::vector<int> v{ 3, 6 };     // v becomes {3, 6}
```

```
// Different from std::vector<int> v{3, 6} in C++11
std::vector<int> v(3, 6);  // v becomes {6, 6, 6}

std::vector<int> v(4);     // v becomes {0, 0, 0, 0}
```

Ein Vektor kann auf verschiedene Weise aus einem anderen Container initialisiert werden:

Kopieren Sie die Konstruktion (nur von einem anderen Vektor), um Daten aus `v2` kopieren:

```
std::vector<int> v(v2);
```

```
std::vector<int> v = v2;
```

C ++ 11

Verschiebe die Konstruktion (nur von einem anderen Vektor), wodurch Daten von `v2` :

```
std::vector<int> v(std::move(v2));
std::vector<int> v = std::move(v2);
```

Iterator (Bereich) Kopierkonstruktion, die Elemente in `v` kopiert:

```
// from another vector
std::vector<int> v(v2.begin(), v2.begin() + 3); // v becomes {v2[0], v2[1], v2[2]}

// from an array
int z[] = { 1, 2, 3, 4 };
std::vector<int> v(z, z + 3);                   // v becomes {1, 2, 3}

// from a list
std::list<int> list1{ 1, 2, 3 };
std::vector<int> v(list1.begin(), list1.end()); // v becomes {1, 2, 3}
```

C ++ 11

Iterator-Move-Konstruktion mithilfe von `std::make_move_iterator` , wodurch Elemente in `v` `std::make_move_iterator` :

```
// from another vector
std::vector<int> v(std::make_move_iterator(v2.begin()),
                   std::make_move_iterator(v2.end());

// from a list
std::list<int> list1{ 1, 2, 3 };
std::vector<int> v(std::make_move_iterator(list1.begin()),
                   std::make_move_iterator(list1.end()));
```

Ein `std::vector` kann nach seiner Konstruktion mit Hilfe der Mitgliedsfunktion `assign()` erneut initialisiert werden:

```
v.assign(4, 100);                     // v becomes {100, 100, 100, 100}

v.assign(v2.begin(), v2.begin() + 3);  // v becomes {v2[0], v2[1], v2[2]}

int z[] = { 1, 2, 3, 4 };
v.assign(z + 1, z + 4);               // v becomes {2, 3, 4}
```

Elemente einfügen

Ein Element am Ende eines Vektors anhängen (durch Kopieren / Verschieben):

```
struct Point {
  double x, y;
  Point(double x, double y) : x(x), y(y) {}
```

```
};
std::vector<Point> v;
Point p(10.0, 2.0);
v.push_back(p);  // p is copied into the vector.
```

C ++ 11

Anhängen eines Elements am Ende eines Vektors durch Konstruieren des Elements an Ort und Stelle:

```
std::vector<Point> v;
v.emplace_back(10.0, 2.0); // The arguments are passed to the constructor of the
                           // given type (here Point). The object is constructed
                           // in the vector, avoiding a copy.
```

Beachten Sie, dass `std::vector` Performancegründen *keine* `push_front()` . Durch das Hinzufügen eines Elements am Anfang werden alle im Vektor vorhandenen Elemente verschoben. Wenn Sie häufig Elemente am Anfang Ihres Containers einfügen möchten, können Sie stattdessen `std::list` oder `std::deque` verwenden.

Einfügen eines Elements an einer beliebigen Position eines Vektors:

```
std::vector<int> v{ 1, 2, 3 };
v.insert(v.begin(), 9);          // v now contains {9, 1, 2, 3}
```

C ++ 11

Einfügen eines Elements an einer beliebigen Position eines Vektors durch Konstruieren des Elements an Ort und Stelle:

```
std::vector<int> v{ 1, 2, 3 };
v.emplace(v.begin()+1, 9);     // v now contains {1, 9, 2, 3}
```

Einfügen eines anderen Vektors an einer beliebigen Position des Vektors:

```
std::vector<int> v(4);      // contains: 0, 0, 0, 0
std::vector<int> v2(2, 10); // contains: 10, 10
v.insert(v.begin()+2, v2.begin(), v2.end()); // contains: 0, 0, 10, 10, 0, 0
```

Einfügen eines Arrays an einer beliebigen Position eines Vektors:

```
std::vector<int> v(4); // contains: 0, 0, 0, 0
int a [] = {1, 2, 3}; // contains: 1, 2, 3
v.insert(v.begin()+1, a, a+sizeof(a)/sizeof(a[0])); // contains: 0, 1, 2, 3, 0, 0, 0
```

Verwenden Sie `reserve()` bevor Sie mehrere Elemente einfügen, wenn die resultierende Vektorgröße vorher bekannt ist, um Mehrfachzuweisungen zu vermeiden (siehe Vektorgröße und -

kapazität):

```
std::vector<int> v;
v.reserve(100);
for(int i = 0; i < 100; ++i)
    v.emplace_back(i);
```

Stellen Sie sicher, dass Sie in diesem Fall nicht den Fehler machen, `resize()` aufzurufen. `resize()` erstellen Sie versehentlich einen Vektor mit 200 Elementen, bei dem nur die letzten 100 den gewünschten Wert haben.

Iteration über std :: vector

Sie können einen `std::vector` auf verschiedene Arten durchlaufen. Für jeden der folgenden Abschnitte ist `v` wie folgt definiert:

```
std::vector<int> v;
```

Iteration in Vorwärtsrichtung

C ++ 11

```
// Range based for
for(const auto& value: v) {
    std::cout << value << "\n";
}

// Using a for loop with iterator
for(auto it = std::begin(v); it != std::end(v); ++it) {
    std::cout << *it << "\n";
}

// Using for_each algorithm, using a function or functor:
void fun(int const& value) {
    std::cout << value << "\n";
}

std::for_each(std::begin(v), std::end(v), fun);

// Using for_each algorithm. Using a lambda:
std::for_each(std::begin(v), std::end(v), [](int const& value) {
    std::cout << value << "\n";
});
```

C ++ 11

```
// Using a for loop with iterator
for(std::vector<int>::iterator it = std::begin(v); it != std::end(v); ++it) {
    std::cout << *it << "\n";
}
```

```
// Using a for loop with index
```

```
for(std::size_t i = 0; i < v.size(); ++i) {
    std::cout << v[i] << "\n";
}
```

Iteration in umgekehrter Richtung

C ++ 14

```
// There is no standard way to use range based for for this.
// See below for alternatives.

// Using for_each algorithm
// Note: Using a lambda for clarity. But a function or functor will work
std::for_each(std::rbegin(v), std::rend(v), [](auto const& value) {
    std::cout << value << "\n";
});

// Using a for loop with iterator
for(auto rit = std::rbegin(v); rit != std::rend(v); ++rit) {
    std::cout << *rit << "\n";
}
```

```
// Using a for loop with index
for(std::size_t i = 0; i < v.size(); ++i) {
    std::cout << v[v.size() - 1 - i] << "\n";
}
```

Es gibt zwar keine integrierte Methode, um den Bereich für die Iteration umzukehren. es ist relativ einfach, dies zu beheben. Der Bereich für die Verwendungszwecke `begin()` und `end()` , um Iteratoren zu erhalten, und die Simulation dieser Werte mit einem Wrapper-Objekt kann die von uns benötigten Ergebnisse erzielen.

C ++ 14

```
template<class C>
struct ReverseRange {
  C c; // could be a reference or a copy, if the original was a temporary
  ReverseRange(C&& cin): c(std::forward<C>(cin)) {}
  ReverseRange(ReverseRange&&)=default;
  ReverseRange& operator=(ReverseRange&&)=delete;
  auto begin() const {return std::rbegin(c);}
  auto end()   const {return std::rend(c);}
};
// C is meant to be deduced, and perfect forwarded into
template<class C>
ReverseRange<C> make_ReverseRange(C&& c) {return {std::forward<C>(c)};}

int main() {
    std::vector<int> v { 1,2,3,4};
    for(auto const& value: make_ReverseRange(v)) {
        std::cout << value << "\n";
    }
}
```

Erzwingen von const-Elementen

Seit C ++ 11 können Sie mit den `cbegin()` und `cend()` einen *konstanten Iterator* für einen Vektor erhalten, auch wenn der Vektor nicht *konstant* ist. Mit einem konstanten Iterator können Sie den Inhalt des Vektors lesen, aber nicht modifizieren. Dies ist nützlich, um die const-Korrektheit zu erzwingen:

C ++ 11

```
// forward iteration
for (auto pos = v.cbegin(); pos != v.cend(); ++pos) {
   // type of pos is vector<T>::const_iterator
   // *pos = 5; // Compile error - can't write via const iterator
}

// reverse iteration
for (auto pos = v.crbegin(); pos != v.crend(); ++pos) {
   // type of pos is vector<T>::const_iterator
   // *pos = 5; // Compile error - can't write via const iterator
}

// expects Functor::operand()(T&)
for_each(v.begin(), v.end(), Functor());

// expects Functor::operand()(const T&)
for_each(v.cbegin(), v.cend(), Functor())
```

C ++ 17

as_const erweitert dies auf die Bereichsiteration:

```
for (auto const& e : std::as_const(v)) {
  std::cout << e << '\n';
}
```

Dies ist in früheren Versionen von C ++ leicht zu implementieren:

C ++ 14

```
template <class T>
constexpr std::add_const_t<T>& as_const(T& t) noexcept {
  return t;
}
```

Ein Hinweis zur Effizienz

Da die Klasse `std::vector` im Grunde eine Klasse ist, die ein dynamisch zugewiesenes zusammenhängendes Array verwaltet, gilt das gleiche Prinzip, das hier erläutert wird , auch für C ++ - Vektoren. Der Zugriff auf den Inhalt des Vektors nach Index ist viel effizienter, wenn das Prinzip der Reihenhauptordnung angewendet wird. Natürlich fügt jeder Zugriff auf den Vektor auch

seinen Verwaltungsinhalt in den Cache ein. Wie jedoch oft diskutiert wurde (insbesondere hier und hier), ist der Leistungsunterschied beim Durchlaufen eines `std::vector` Vergleich zu einem rohen Array Ist vernachlässigbar. Das gleiche Prinzip der Effizienz für unformatierte Arrays in C gilt also auch für C ++ s `std::vector` .

Zugriff auf Elemente

Es gibt zwei Möglichkeiten, auf Elemente in einem `std::vector` zuzugreifen

- Indexbasierter Zugriff
- Iteratoren

Indexbasierter Zugriff:

Dies kann entweder mit dem Indexoperator `[]` oder der Memberfunktion `at()` .

Beide geben an der entsprechenden Stelle im `std::vector` einen Verweis auf das Element zurück (es sei denn, es handelt sich um einen `vector<bool>`), sodass es sowohl gelesen als auch modifiziert werden kann (wenn der Vektor nicht `const`).

`[]` und `at()` unterscheiden sich dahingehend, dass `[]` nicht garantiert wird, dass Grenzen geprüft werden, während `at()` tut. Der Zugriff auf Elemente, bei denen `index < 0` oder `index >= size` ist, ist für `[]` definiertes Verhalten , während `at()` eine Ausnahme `std::out_of_range` .

Anmerkung: In den folgenden Beispielen wird zur Vereinfachung die Initialisierung im C ++ 11-Stil verwendet. Die Operatoren können jedoch mit allen Versionen verwendet werden (sofern nicht C ++ 11 markiert ist).

C ++ 11

```
std::vector<int> v{ 1, 2, 3 };
```

```
// using []
int a = v[1];    // a is 2
v[1] = 4;        // v now contains { 1, 4, 3 }

// using at()
int b = v.at(2); // b is 3
v.at(2) = 5;     // v now contains { 1, 4, 5 }
int c = v.at(3); // throws std::out_of_range exception
```

Da die `at()` Methode eine Begrenzungsprüfung durchführt und Ausnahmen auslösen kann, ist sie langsamer als `[]` . Dies macht `[]` bevorzugten Code, bei dem die Semantik der Operation gewährleistet, dass der Index in Grenzen ist. In jedem Fall erfolgen Zugriffe auf Elemente von Vektoren in konstanter Zeit. Das heißt, der Zugriff auf das erste Element des Vektors hat (zeitlich) die gleichen Kosten für den Zugriff auf das zweite Element, das dritte Element usw.

Betrachten Sie zum Beispiel diese Schleife

```
for (std::size_t i = 0; i < v.size(); ++i) {
    v[i] = 1;
}
```

Hier wissen wir, dass die Indexvariable `i` immer in Grenzen ist. Es wäre also eine Verschwendung von CPU-Zyklen, zu prüfen, ob `i` für jeden Aufruf an `operator[]` in Grenzen ist.

Die Elementfunktionen `front()` und `back()` ermöglichen einen einfachen Referenzzugriff auf das erste bzw. letzte Element des Vektors. Diese Positionen werden häufig verwendet, und die speziellen Zugriffsmethoden können mit `[]` besser gelesen werden als ihre Alternativen:

```
std::vector<int> v{ 4, 5, 6 }; // In pre-C++11 this is more verbose

int a = v.front();   // a is 4, v.front() is equivalent to v[0]
v.front() = 3;       // v now contains {3, 5, 6}
int b = v.back();    // b is 6, v.back() is equivalent to v[v.size() - 1]
v.back() = 7;        // v now contains {3, 5, 7}
```

Hinweis : Das Aufrufen von `front()` oder `back()` auf einem leeren Vektor ist undefiniert . Sie müssen vor dem Aufruf von `front()` oder `back()` überprüfen, ob der Container nicht leer ist, indem Sie die `empty()` Member-Funktion (die prüft, ob der Container leer ist) anzeigen. Ein einfaches Beispiel für die Verwendung von 'empty ()' zum Testen auf einen leeren Vektor folgt:

```
int main ()
{
  std::vector<int> v;
  int sum (0);

  for (int i=1;i<=10;i++) v.push_back(i);//create and initialize the vector

  while (!v.empty())//loop through until the vector tests to be empty
  {
     sum += v.back();//keep a running total
     v.pop_back();//pop out the element which removes it from the vector
  }

  std::cout << "total: " << sum << '\n';//output the total to the user

  return 0;
}
```

Im obigen Beispiel wird ein Vektor mit einer Zahlenfolge von 1 bis 10 erstellt. Anschließend werden die Elemente des Vektors herausgefahren, bis der Vektor leer ist (mithilfe von 'empty ()'), um undefiniertes Verhalten zu verhindern. Dann wird die Summe der Zahlen im Vektor berechnet und dem Benutzer angezeigt.

C ++ 11

Die `data()` Methode gibt einen Zeiger auf den Rohspeicher zurück, der vom `std::vector` , um seine Elemente intern zu speichern. Dies wird am häufigsten verwendet, wenn die Vektordaten an älteren Code übergeben werden, der ein Array im C-Stil erwartet.

```
std::vector<int> v{ 1, 2, 3, 4 }; // v contains {1, 2, 3, 4}
```

```
int* p = v.data(); // p points to 1
*p = 4;            // v now contains {4, 2, 3, 4}
++p;               // p points to 2
*p = 3;            // v now contains {4, 3, 3, 4}
p[1] = 2;          // v now contains {4, 3, 2, 4}
*(p + 2) = 1;      // v now contains {4, 3, 2, 1}
```

C ++ 11

Vor C ++ 11 kann die `data()` Methode simuliert werden, indem `front()` aufgerufen wird und die Adresse des zurückgegebenen Werts verwendet wird:

```
std::vector<int> v(4);
int* ptr = &(v.front()); // or &v[0]
```

Dies funktioniert, weil Vektoren immer garantiert sind, dass ihre Elemente an benachbarten Speicherorten gespeichert werden, vorausgesetzt, der Inhalt des Vektors überschreibt nicht den unären `operator&` . In diesem Fall müssen Sie `std::addressof` in Pre-C ++ 11 erneut implementieren. Es wird auch davon ausgegangen, dass der Vektor nicht leer ist.

Iteratoren:

Iteratoren werden im Beispiel "Iterieren über `std::vector` " und im Artikel Iteratoren näher erläutert. Kurz gesagt, sie verhalten sich ähnlich wie Zeiger auf die Elemente des Vektors:

C ++ 11

```
std::vector<int> v{ 4, 5, 6 };

auto it = v.begin();
int i = *it;         // i is 4
++it;
i = *it;             // i is 5
*it = 6;             // v contains { 4, 6, 6 }
auto e = v.end();    // e points to the element after the end of v. It can be
                     // used to check whether an iterator reached the end of the vector:
++it;
it == v.end();       // false, it points to the element at position 2 (with value 6)
++it;
it == v.end();       // true
```

Er steht im Einklang mit dem Standard , dass ein `std::vector<T>` ,s Iteratoren tatsächlich *sein* `T*` s, aber die meisten Standard - Bibliotheken tun dies nicht. Wenn Sie dies nicht tun, werden sowohl Fehlermeldungen verbessert als auch nicht portabler Code abgerufen. Außerdem können Sie die Iteratoren mit Debugging-Überprüfungen in Builds ohne Releases versehen. In Release-Builds wird der Klassenumbruch um den zugrundeliegenden Zeiger entfernt.

Sie können eine Referenz oder einen Zeiger auf ein Element eines Vektors für den indirekten Zugriff beibehalten. Diese Verweise oder Zeiger auf Elemente im `vector` bleiben stabil und der Zugriff bleibt definiert, es sei denn, Sie fügen Elemente am oder vor dem Element im `vector` oder

entfernen die `vector` . Dies entspricht der Regel für das Ungültigmachen von Iteratoren.

C ++ 11

```
std::vector<int> v{ 1, 2, 3 };
int* p = v.data() + 1;     // p points to 2
v.insert(v.begin(), 0);    // p is now invalid, accessing *p is a undefined behavior.
p = v.data() + 1;          // p points to 1
v.reserve(10);             // p is now invalid, accessing *p is a undefined behavior.
p = v.data() + 1;          // p points to 1
v.erase(v.begin());        // p is now invalid, accessing *p is a undefined behavior.
```

Verwendung von std :: vector als C-Array

Es gibt mehrere Möglichkeiten, einen `std::vector` als C-Array zu verwenden (z. B. zur Kompatibilität mit C-Bibliotheken). Dies ist möglich, weil die Elemente in einem Vektor zusammenhängend gespeichert werden.

C ++ 11

```
std::vector<int> v{ 1, 2, 3 };
int* p = v.data();
```

Im Gegensatz zu Lösungen, die auf früheren C ++ - Standards basieren (siehe unten), kann die `.data()` auch auf leere Vektoren angewendet werden, da sie in diesem Fall kein undefiniertes Verhalten verursacht.

Vor C ++ 11 müssten Sie die Adresse des ersten Elements des Vektors verwenden, um einen entsprechenden Zeiger zu erhalten. Wenn der Vektor nicht leer ist, sind diese beiden Methoden austauschbar:

```
int* p = &v[0];      // combine subscript operator and 0 literal

int* p = &v.front(); // explicitly reference the first element
```

Hinweis: Wenn der Vektor leer ist, sind `v[0]` und `v.front()` nicht definiert und können nicht verwendet werden.

Wenn Sie die Basisadresse der Vektordaten speichern, beachten Sie, dass viele Operationen (wie `push_back` , `resize` usw.) den Datenspeicherort des Vektors `resize` können, wodurch vorherige Datenzeiger ungültig werden . Zum Beispiel:

```
std::vector<int> v;
int* p = v.data();
v.resize(42);      // internal memory location changed; value of p is now invalid
```

Iterator / Zeiger-Invalidierung

Iteratoren und Zeiger, die auf einen `std::vector` können nur bei bestimmten Operationen ungültig werden. Die Verwendung ungültiger Iteratoren / Zeiger führt zu undefiniertem Verhalten.

Zu den Operationen, die Iteratoren / Zeiger ungültig machen, gehören:

- Jeder Einfügevorgang, der die `capacity` des `vector` ändert, macht *alle* Iteratoren / Zeiger ungültig:

```
vector<int> v(5); // Vector has a size of 5; capacity is unknown.
int *p1 = &v[0];
v.push_back(2);   // p1 may have been invalidated, since the capacity was unknown.

v.reserve(20);    // Capacity is now at least 20.
int *p2 = &v[0];
v.push_back(4);   // p2 is *not* invalidated, since the size of `v` is now 7.
v.insert(v.end(), 30, 9); // Inserts 30 elements at the end. The size exceeds the
                          // requested capacity of 20, so `p2` is (probably) invalidated.
int *p3 = &v[0];
v.reserve(v.capacity() + 20); // Capacity exceeded, thus `p3` is invalid.
```

C ++ 11

```
auto old_cap = v.capacity();
v.shrink_to_fit();
if(old_cap != v.capacity())
    // Iterators were invalidated.
```

- Jeder Einfügevorgang, der die Kapazität nicht erhöht, macht die Iteratoren / Zeiger, die auf Elemente an der Einfügeposition zeigen, über diese hinaus ungültig. Dazu gehört auch das `end` Iterator:

```
vector<int> v(5);
v.reserve(20);                 // Capacity is at least 20.
int *p1 = &v[0];
int *p2 = &v[3];
v.insert(v.begin() + 2, 5, 0); // `p2` is invalidated, but since the capacity
                               // did not change, `p1` remains valid.
int *p3 = &v[v.size() - 1];
v.push_back(10); // The capacity did not change, so `p3` and `p1` remain valid.
```

- Bei jedem Entfernungsvorgang werden Iteratoren / Zeiger ungültig, die auf die entfernten Elemente und auf alle Elemente nach den entfernten Elementen zeigen. Dazu gehört auch das `end` Iterator:

```
vector<int> v(10);
int *p1 = &v[0];
int *p2 = &v[5];
v.erase(v.begin() + 3, v.end()); // `p2` is invalid, but `p1` remains valid.
```

- `operator=` (Kopieren, Verschieben oder anderweitig) und `clear()` alle Iteratoren / Zeiger ungültig, die in den Vektor zeigen.

Elemente löschen

Das letzte Element löschen:

```
std::vector<int> v{ 1, 2, 3 };
v.pop_back();                           // v becomes {1, 2}
```

Alle Elemente löschen:

```
std::vector<int> v{ 1, 2, 3 };
v.clear();                              // v becomes an empty vector
```

Element nach Index löschen:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(v.begin() + 3);                 // v becomes {1, 2, 3, 5, 6}
```

Hinweis: Für einen `vector` ein Element zu löschen , die nicht das letzte Element ist, alle Elemente über das gelöschte Element werden müssen kopiert oder verschoben , die Lücke zu füllen, siehe Hinweis unten und std :: list .

Alle Elemente eines Bereichs löschen:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(v.begin() + 1, v.begin() + 5);  // v becomes {1, 6}
```

Hinweis: Die oben genannten Methoden ändern nicht die Kapazität des Vektors, sondern nur die Größe. Siehe Vektorgröße und Kapazität .

Die `erase` , die eine Reihe von Elementen entfernt, wird häufig als Teil des **Lösch-Entfernungs-**Idioms verwendet. Das heißt, zuerst `std::remove` verschiebt einige Elemente an das Ende des Vektors und `erase` sie dann. Dies ist eine relativ ineffiziente Operation für alle Indizes, die unter dem letzten Index des Vektors liegen, da alle Elemente nach den gelöschten Segmenten an neue Positionen verschoben werden müssen. Geschwindigkeitskritische Anwendungen, die das effiziente Entfernen beliebiger Elemente in einem Container erfordern, finden Sie unter std :: list .

Elemente nach Wert löschen:

```
std::vector<int> v{ 1, 1, 2, 2, 3, 3 };
int value_to_remove = 2;
v.erase(std::remove(v.begin(), v.end(), value_to_remove), v.end()); // v becomes {1, 1, 3, 3}
```

Elemente nach Bedingung löschen:

```
// std::remove_if needs a function, that takes a vector element as argument and returns true,
// if the element shall be removed
bool _predicate(const int& element) {
    return (element > 3); // This will cause all elements to be deleted that are larger than 3
}
...
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(std::remove_if(v.begin(), v.end(), _predicate), v.end()); // v becomes {1, 2, 3}
```

Löschen von Elementen durch Lambda, ohne zusätzliche Prädikatsfunktion zu erstellen

C ++ 11

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
v.erase(std::remove_if(v.begin(), v.end(),
     [](auto& element){return element > 3;} ), v.end()
);
```

Elemente nach Bedingung aus einer Schleife löschen:

```
std::vector<int> v{ 1, 2, 3, 4, 5, 6 };
std::vector<int>::iterator it = v.begin();
while (it != v.end()) {
    if (condition)
        it = v.erase(it); // after erasing, 'it' will be set to the next element in v
    else
        ++it;             // manually set 'it' to the next element in v
}
```

Es ist zwar wichtig, dass Sie `it` im Falle eines Löschvorgangs *nicht* inkrementieren, Sie sollten jedoch eine andere Methode verwenden, wenn Sie sie dann wiederholt in einer Schleife löschen. `remove_if` Sie `remove_if` für einen effizienteren Weg.

Elemente nach Bedingung aus einer umgekehrten Schleife löschen:

```
std::vector<int> v{ -1, 0, 1, 2, 3, 4, 5, 6 };
typedef std::vector<int>::reverse_iterator rev_itr;
```

```
rev_itr it = v.rbegin();

while (it != v.rend()) { // after the loop only '0' will be in v
    int value = *it;
    if (value) {
        ++it;
        // See explanation below for the following line.
        it = rev_itr(v.erase(it.base()));
    } else
        ++it;
}
```

Beachten Sie einige Punkte für die vorhergehende Schleife:

- Bei einem Reverse - Iterator `it` zu einem Element zeigt, das Verfahren `base` gibt dem regulären (nicht umgekehrt) Iterator zeigt auf das gleiche Element.
- `vector::erase(iterator)` löscht das Element, auf das ein Iterator zeigt, und gibt einen Iterator an das Element zurück, das dem angegebenen Element folgte.
- `reverse_iterator::reverse_iterator(iterator)` einen Reverse-Iterator aus einem Iterator.

Setzen Sie insgesamt die Linie `it = rev_itr(v.erase(it.base()))` sagt: Nehmen Sie die Reverse - Iterator `it` haben `v` Löschen Sie das Element durch seine regelmäßigen Iterator zugespitzt; nehmen die resultierende Iterator, eine umgekehrte Iterator daraus, baut und Zuweisen zu dem umgekehrten Iterator `it` .

Durch das Löschen aller Elemente mit `v.clear()` wird kein Speicherplatz frei (`capacity()` des Vektors bleibt unverändert). Um Platz zu gewinnen, verwenden Sie:

```
std::vector<int>().swap(v);
```

C ++ 11

`shrink_to_fit()` ungenutzte Vektorkapazität frei:

```
v.shrink_to_fit();
```

Das `shrink_to_fit` garantiert zwar nicht wirklich Speicherplatz zurückzugewinnen, aber die meisten aktuellen Implementierungen tun dies.

Ein Element in std :: vector finden

Mit der Funktion `std::find` , definiert im Header `<algorithm>` , kann ein Element in einem `std::vector` .

`std::find` verwendet den `operator==` , um Elemente auf Gleichheit zu vergleichen. Es gibt einen Iterator an das erste Element in dem Bereich zurück, der dem Wert gleich ist.

Wenn das betreffende Element nicht gefunden wird, gibt `std::find` `std::vector::end` (oder

`std::vector::cend` wenn der Vektor `const`).

C ++ 11

```
static const int arr[] = {5, 4, 3, 2, 1};
std::vector<int> v (arr, arr + sizeof(arr) / sizeof(arr[0]) );

std::vector<int>::iterator it = std::find(v.begin(), v.end(), 4);
std::vector<int>::difference_type index = std::distance(v.begin(), it);
// `it` points to the second element of the vector, `index` is 1

std::vector<int>::iterator missing = std::find(v.begin(), v.end(), 10);
std::vector<int>::difference_type index_missing = std::distance(v.begin(), missing);
// `missing` is v.end(), `index_missing` is 5 (ie. size of the vector)
```

C ++ 11

```
std::vector<int> v { 5, 4, 3, 2, 1 };

auto it = std::find(v.begin(), v.end(), 4);
auto index = std::distance(v.begin(), it);
// `it` points to the second element of the vector, `index` is 1

auto missing = std::find(v.begin(), v.end(), 10);
auto index_missing = std::distance(v.begin(), missing);
// `missing` is v.end(), `index_missing` is 5 (ie. size of the vector)
```

Wenn Sie viele Suchvorgänge in einem großen Vektor durchführen müssen, sollten Sie den Vektor zuerst sortieren, bevor Sie den `binary_search` Algorithmus verwenden.

Um das erste Element in einem Vektor zu finden, der eine Bedingung erfüllt, kann `std::find_if` verwendet werden. Zusätzlich zu den zwei für `std::find` angegebenen Parametern akzeptiert `std::find_if` ein drittes Argument, das ein Funktionsobjekt oder einen Funktionszeiger auf eine Prädikatfunktion ist. Das Prädikat sollte ein Element aus dem Container als Argument akzeptieren und einen in `bool` konvertierbaren Wert zurückgeben, ohne den Container zu `bool` :

C ++ 11

```
bool isEven(int val) {
    return (val % 2 == 0);
}

struct moreThan {
    moreThan(int limit) : _limit(limit) {}

    bool operator()(int val) {
        return val > _limit;
    }

    int _limit;
};

static const int arr[] = {1, 3, 7, 8};
std::vector<int> v (arr, arr + sizeof(arr) / sizeof(arr[0]) );
```

```
std::vector<int>::iterator it = std::find_if(v.begin(), v.end(), isEven);
// `it` points to 8, the first even element

std::vector<int>::iterator missing = std::find_if(v.begin(), v.end(), moreThan(10));
// `missing` is v.end(), as no element is greater than 10
```

C ++ 11

```
// find the first value that is even
std::vector<int> v = {1, 3, 7, 8};
auto it = std::find_if(v.begin(), v.end(), [](int val){return val % 2 == 0;});
// `it` points to 8, the first even element

auto missing = std::find_if(v.begin(), v.end(), [](int val){return val > 10;});
// `missing` is v.end(), as no element is greater than 10
```

Konvertieren eines Arrays in std :: vector

Ein Array kann einfach mit Hilfe von `std::begin` und `std::end` in einen `std::vector` umgewandelt werden:

C ++ 11

```
int values[5] = { 1, 2, 3, 4, 5 }; // source array

std::vector<int> v(std::begin(values), std::end(values)); // copy array to new vector

for(auto &x: v)
    std::cout << x << " ";
std::cout << std::endl;
```

1 2 3 4 5

```
int main(int argc, char* argv[]) {
    // convert main arguments into a vector of strings.
    std::vector<std::string>  args(argv, argv + argc);
}
```

Eine C ++ 11-Initialisierungsliste <> kann auch verwendet werden, um den Vektor sofort zu initialisieren

```
initializer_list<int> arr = { 1,2,3,4,5 };
vector<int> vec1 {arr};

for (auto & i : vec1)
    cout << i << endl;
```

Vektor : Die Ausnahme zu so vielen Regeln

Der Standard (Abschnitt 23.3.7) gibt an, dass eine Spezialisierung des `vector<bool>` vorgesehen ist, die den Speicherplatz optimiert, indem die `bool` Werte `bool` werden, sodass jeder nur ein Bit beansprucht. Da Bits in C ++ nicht adressierbar sind, bedeutet dies, dass einige Anforderungen an

den `vector` nicht auf den `vector<bool>` :

- Die gespeicherten Daten müssen nicht zusammenhängend sein, sodass ein `vector<bool>` nicht an eine C-API übergeben werden kann, die ein `bool` Array erwartet.
- `at()` , `operator []` und Dereferenzierung von Iteratoren geben keine Referenz auf `bool` . Sie geben stattdessen ein Proxy-Objekt zurück, das (unvollkommen) einen Verweis auf einen `bool` simuliert, indem seine Zuweisungsoperatoren überladen werden. Der folgende Code ist beispielsweise für `std::vector<bool>` möglicherweise nicht gültig, da die Dereferenzierung eines Iterators keine Referenz zurückgibt:

C ++ 11

```
std::vector<bool> v = {true, false};
for (auto &b: v) { } // error
```

Ebenso können Funktionen, die ein `bool&` argument erwarten, nicht mit dem Ergebnis des `operator []` oder `at()` auf den `vector<bool>` oder mit dem Ergebnis der Dereferenzierung des Iterators verwendet werden:

```
  void f(bool& b);
  f(v[0]);             // error
  f(*v.begin());       // error
```

Die Implementierung von `std::vector<bool>` hängt sowohl vom Compiler als auch von der Architektur ab. Die Spezialisierung wird implementiert, indem `n` Booleans in den untersten adressierbaren Speicherbereich gepackt werden. Hier ist `n` die Größe des niedrigsten adressierbaren Speichers in Bits. In den meisten modernen Systemen sind dies 1 Byte oder 8 Bit. Das bedeutet, dass ein Byte 8 boolesche Werte speichern kann. Dies ist eine Verbesserung gegenüber der herkömmlichen Implementierung, bei der ein boolescher Wert in 1 Byte Speicher gespeichert wird.

Hinweis: Das folgende Beispiel zeigt mögliche bitweise Werte einzelner Bytes in einem traditionellen vs. optimierten `vector<bool>` . Dies gilt nicht immer für alle Architekturen. Es ist jedoch eine gute Möglichkeit, die Optimierung zu visualisieren. In den folgenden Beispielen wird ein Byte als [x, x, x, x, x, x, x, x] dargestellt.

Traditionelles *`std::vector<char>` speichert 8 boolesche Werte:*

C ++ 11

```
std::vector<char> trad_vect = {true, false, false, false, true, false, true, true};
```

Bitweise Darstellung:

```
[0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,0],
[0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,0], [0,0,0,0,0,0,0,1], [0,0,0,0,0,0,0,1]
```

Specialized *`std::vector<bool>` speichert 8 boolesche Werte:*

C ++ 11

```
std::vector<bool> optimized_vect = {true, false, false, false, true, false, true, true};
```

Bitweise Darstellung:

```
[1,0,0,0,1,0,1,1]
```

Beachten Sie im obigen Beispiel, dass in der traditionellen Version von `std::vector<bool>` 8 boolesche Werte 8 Byte Speicherplatz beanspruchen, wohingegen in der optimierten Version von `std::vector<bool>` nur 1 Byte verwendet wird Erinnerung. Dies ist eine deutliche Verbesserung der Speicherauslastung. Wenn Sie einen `vector<bool>` an eine API im C-Stil übergeben müssen, müssen Sie die Werte möglicherweise in ein Array kopieren oder einen besseren Weg finden, um die API zu verwenden, wenn Arbeitsspeicher und Leistung gefährdet sind.

Vektorgröße und Kapazität

Vektorgröße ist einfach die Anzahl der Elemente im Vektor:

1. Aktuelle **Vektorgröße** wird durch abgefragten `size()` Member - Funktion. Die Funktion `empty()` gibt `true` zurück `true` wenn size 0 ist:

   ```
   vector<int> v = { 1, 2, 3 }; // size is 3
   const vector<int>::size_type size = v.size();
   cout << size << endl; // prints 3
   cout << boolalpha << v.empty() << endl; // prints false
   ```

2. Der standardmäßig konstruierte Vektor beginnt mit einer Größe von 0:

   ```
   vector<int> v; // size is 0
   cout << v.size() << endl; // prints 0
   ```

3. Durch Hinzufügen von `N` Elementen zum Vektor wird die **Größe** um `N` erhöht (z. B. durch die Funktionen `push_back()` , `insert()` oder `resize()`).

4. Durch das Entfernen von `N` Elementen aus dem Vektor wird die **Größe** um `N` verringert (z. B. durch `pop_back()` , `erase()` oder `clear()` Funktionen).

5. Vector hat eine implementierungsspezifische Obergrenze für die Größe, aber es ist wahrscheinlich, dass der RAM knapp wird, bevor er erreicht wird:

   ```
   vector<int> v;
   const vector<int>::size_type max_size = v.max_size();
   cout << max_size << endl; // prints some large number
   v.resize( max_size ); // probably won't work
   v.push_back( 1 ); // definitely won't work
   ```

Häufiger Fehler: **Größe** ist nicht notwendigerweise (oder sogar normalerweise) `int` :

```
// !!!bad!!!evil!!!
vector<int> v_bad( N, 1 ); // constructs large N size vector
```

```
for( int i = 0; i < v_bad.size(); ++i ) { // size is not supposed to be int!
    do_something( v_bad[i] );
}
```

Die Vektorkapazität unterscheidet sich von der **Größe** . Während **size** einfach ist, wie viele Elemente der Vektor derzeit hat, ist **Kapazität** für wie viele Elemente Speicherplatz reserviert / reserviert. Das ist nützlich, weil zu häufige (Neu-) Zuweisungen zu großer Größen teuer sein können.

1. Stromvektor **Kapazität** wird durch abgefragt `capacity()` Mitgliedsfunktion. **Die Kapazität** ist immer größer oder gleich der **Größe** :

   ```
   vector<int> v = { 1, 2, 3 }; // size is 3, capacity is >= 3
   const vector<int>::size_type capacity = v.capacity();
   cout << capacity << endl; // prints number >= 3
   ```

2. Sie können Kapazität manuell durch `reserve( N )` -Funktion `reserve( N )` Vektorkapazität wird in `N` geändert):

   ```
   // !!!bad!!!evil!!!
   vector<int> v_bad;
   for( int i = 0; i < 10000; ++i ) {
       v_bad.push_back( i ); // possibly lot of reallocations
   }

   // good
   vector<int> v_good;
   v_good.reserve( 10000 ); // good! only one allocation
   for( int i = 0; i < 10000; ++i ) {
       v_good.push_back( i ); // no allocations needed anymore
   }
   ```

3. Sie können anfordern, dass die überschüssige Kapazität von `shrink_to_fit()` freigegeben wird (aber die Implementierung muss Ihnen nicht gehorchen). Dies ist nützlich, um verwendeten Speicher zu sparen:

   ```
   vector<int> v = { 1, 2, 3, 4, 5 }; // size is 5, assume capacity is 6
   v.shrink_to_fit(); // capacity is 5 (or possibly still 6)
   cout << boolalpha << v.capacity() == v.size() << endl; // prints likely true (but
   possibly false)
   ```

Vector verwaltet die Kapazität zum Teil automatisch, wenn Sie Elemente hinzufügen, entscheidet er sich möglicherweise für eine Vergrößerung. Implementierer verwenden gerne 2 oder 1,5 für den Wachstumsfaktor (Golden Ratio wäre der ideale Wert - ist jedoch aufgrund der rationalen Anzahl nicht praktikabel). Auf der anderen Seite schrumpfen Vektor normalerweise nicht automatisch. Zum Beispiel:

```
vector<int> v; // capacity is possibly (but not guaranteed) to be 0
v.push_back( 1 ); // capacity is some starter value, likely 1
v.clear(); // size is 0 but capacity is still same as before!

v = { 1, 2, 3, 4 }; // size is 4, and lets assume capacity is 4.
```

```
v.push_back( 5 ); // capacity grows - let's assume it grows to 6 (1.5 factor)
v.push_back( 6 ); // no change in capacity
v.push_back( 7 ); // capacity grows - let's assume it grows to 9 (1.5 factor)
// and so on
v.pop_back(); v.pop_back(); v.pop_back(); v.pop_back(); // capacity stays the same
```

Verkettung von Vektoren

Ein `std::vector` kann mithilfe der Member-Funktion `insert()` an einen anderen angehängt werden:

```
std::vector<int> a = {0, 1, 2, 3, 4};
std::vector<int> b = {5, 6, 7, 8, 9};

a.insert(a.end(), b.begin(), b.end());
```

Diese Lösung schlägt jedoch fehl, wenn Sie versuchen, einen Vektor an sich selbst anzuhängen, da der Standard festlegt, dass die für `insert()` angegebenen Iteratoren nicht im selben Bereich liegen dürfen wie die Elemente des Empfängerobjekts.

c ++ 11

Anstelle der Member-Funktionen des Vektors können die Funktionen `std::begin()` und `std::end()` verwendet werden:

```
a.insert(std::end(a), std::begin(b), std::end(b));
```

Dies ist zum Beispiel eine allgemeinere Lösung, da `b` auch ein Array sein kann. Mit dieser Lösung können Sie jedoch keinen Vektor an sich selbst anhängen.

Wenn die Reihenfolge der Elemente im Empfangsvektor keine Rolle spielt, können durch die Berücksichtigung der Anzahl der Elemente in jedem Vektor unnötige Kopiervorgänge vermieden werden:

```
if (b.size() < a.size())
  a.insert(a.end(), b.begin(), b.end());
else
  b.insert(b.end(), a.begin(), a.end());
```

Die Kapazität eines Vektors reduzieren

Ein `std::vector` erhöht nach Bedarf automatisch seine Kapazität beim Einfügen, verringert jedoch niemals seine Kapazität nach dem Entfernen von Elementen.

```
// Initialize a vector with 100 elements
std::vector<int> v(100);

// The vector's capacity is always at least as large as its size
auto const old_capacity = v.capacity();
// old_capacity >= 100

// Remove half of the elements
```

```
v.erase(v.begin() + 50, v.end());  // Reduces the size from 100 to 50 (v.size() == 50),
                                   // but not the capacity (v.capacity() == old_capacity)
```

Um seine Kapazität zu reduzieren, können wir den Inhalt eines Vektors in einen neuen temporären Vektor kopieren. Der neue Vektor hat die minimale Kapazität, die zum Speichern aller Elemente des ursprünglichen Vektors erforderlich ist. Wenn die Größenverringerung des ursprünglichen Vektors signifikant war, ist die Kapazitätsverringerung für den neuen Vektor wahrscheinlich signifikant. Wir können dann den ursprünglichen Vektor mit dem temporären Vektor tauschen, um seine minimierte Kapazität zu erhalten:

```
std::vector<int>(v).swap(v);
```

C ++ 11

In C ++ 11 können wir die `shrink_to_fit()` für einen ähnlichen Effekt verwenden:

```
v.shrink_to_fit();
```

Hinweis: Die `shrink_to_fit()` ist eine Anforderung und garantiert keine Reduzierung der Kapazität.

Verwenden eines sortierten Vektors für die schnelle Elementsuche

Der `<algorithm>` -Header bietet eine Reihe nützlicher Funktionen zum Arbeiten mit sortierten Vektoren.

Eine wichtige Voraussetzung für das Arbeiten mit sortierten Vektoren ist, dass die gespeicherten Werte mit `<` vergleichbar sind.

Ein unsortierter Vektor kann mit der Funktion `std::sort()` sortiert werden:

```
std::vector<int> v;
// add some code here to fill v with some elements
std::sort(v.begin(), v.end());
```

Sortierte Vektoren ermöglichen eine effiziente Elementsuche mit der Funktion `std::lower_bound()` . Im Gegensatz zu `std::find()` führt dies eine effiziente binäre Suche nach dem Vektor durch. Der Nachteil ist, dass es nur für sortierte Eingabebereiche gültige Ergebnisse gibt:

```
// search the vector for the first element with value 42
std::vector<int>::iterator it = std::lower_bound(v.begin(), v.end(), 42);
if (it != v.end() && *it == 42) {
    // we found the element!
}
```

Hinweis: Wenn der angeforderte Wert nicht Teil des Vektors ist, gibt `std::lower_bound()` einen Iterator an das erste Element zurück, das *größer* als der angeforderte Wert ist. Dieses Verhalten ermöglicht es uns, ein neues Element an der richtigen Stelle in einen bereits sortierten Vektor einzufügen:

```
int const new_element = 33;
v.insert(std::lower_bound(v.begin(), v.end(), new_element), new_element);
```

Wenn Sie viele Elemente gleichzeitig einfügen müssen, ist es möglicherweise effizienter, zuerst `push_back()` für alle Elemente und dann `std::sort()` aufzurufen, nachdem alle Elemente eingefügt wurden. In diesem Fall können sich die erhöhten Sortierkosten für die reduzierten Einfügungskosten neuer Elemente am Ende des Vektors und nicht in der Mitte bezahlt machen.

Wenn Ihr Vektor mehrere Elemente mit demselben Wert enthält, versucht `std::lower_bound()` , einen Iterator an das erste Element des gesuchten Werts zurückzugeben. Wenn Sie jedoch *nach* dem letzten Element des gesuchten Werts ein neues Element einfügen müssen, sollten Sie die Funktion `std::upper_bound()` da dies zu einer geringeren Verschiebung der Elemente führt:

```
v.insert(std::upper_bound(v.begin(), v.end(), new_element), new_element);
```

Wenn Sie sowohl den oberen als auch den unteren Iterator benötigen, können Sie die Funktion `std::equal_range()` , um beide effizient mit einem Aufruf abzurufen:

```
std::pair<std::vector<int>::iterator,
          std::vector<int>::iterator> rg = std::equal_range(v.begin(), v.end(), 42);
std::vector<int>::iterator lower_bound = rg.first;
std::vector<int>::iterator upper_bound = rg.second;
```

Um zu testen, ob ein Element in einem sortierten Vektor vorhanden ist (obwohl nicht vektorspezifisch), können Sie die Funktion `std::binary_search()` :

```
bool exists = std::binary_search(v.begin(), v.end(), value_to_find);
```

Funktionen, die große Vektoren zurückgeben

C ++ 11

In C ++ 11 müssen Compiler implizit von einer zurückgegebenen lokalen Variablen verschoben werden. Darüber hinaus können die meisten Compiler in vielen Fällen eine Kopierentscheidung durchführen und die Bewegung komplett ausschalten. Die Rückgabe großer Objekte, die billig bewegt werden können, erfordert daher keine besondere Behandlung mehr:

```
#include <vector>
#include <iostream>

// If the compiler is unable to perform named return value optimization (NRVO)
// and elide the move altogether, it is required to move from v into the return value.
std::vector<int> fillVector(int a, int b) {
    std::vector<int> v;
    v.reserve(b-a+1);
    for (int i = a; i <= b; i++) {
        v.push_back(i);
    }
    return v; // implicit move
}
```

```
int main() { // declare and fill vector
    std::vector<int> vec = fillVector(1, 10);

    // print vector
    for (auto value : vec)
        std::cout << value << " "; // this will print "1 2 3 4 5 6 7 8 9 10 "

    std::cout << std::endl;

    return 0;
}
```

C ++ 11

Vor C ++ 11 wurde die Freigabe von Kopien bereits von den meisten Compilern zugelassen und implementiert. Aufgrund fehlender Semantik der Verschiebung können Sie in älteren Code oder Code, der mit älteren Compilerversionen, die diese Optimierung nicht implementieren, kompiliert werden muss, Vektoren finden, die als Ausgabeargumente übergeben werden, um die nicht benötigte Kopie zu verhindern:

```
#include <vector>
#include <iostream>

// passing a std::vector by reference
void fillVectorFrom_By_Ref(int a, int b, std::vector<int> &v) {
    assert(v.empty());
    v.reserve(b-a+1);
    for (int i = a; i <= b; i++) {
        v.push_back(i);
    }
}

int main() {// declare vector
    std::vector<int> vec;

    // fill vector
    fillVectorFrom_By_Ref(1, 10, vec);
    // print vector
    for (std::vector<int>::const_iterator it = vec.begin(); it != vec.end(); ++it)
        std::cout << *it << " "; // this will print "1 2 3 4 5 6 7 8 9 10 "
    std::cout << std::endl;
    return 0;
}
```

Finden Sie das maximale und minimale Element und den jeweiligen Index in einem Vektor

Um das in einem Vektor gespeicherte größte oder kleinste Element zu finden, können Sie die Methoden `std::max_element` bzw. `std::min_element` verwenden. Diese Methoden sind im Header `<algorithm>` definiert. Wenn mehrere Elemente dem größten (kleinsten) Element entsprechen, geben die Methoden den Iterator an das erste dieser Elemente zurück. Rückgabe `v.end()` für leere Vektoren.

```
std::vector<int> v = {5, 2, 8, 10, 9};
```

```
int maxElementIndex = std::max_element(v.begin(),v.end()) - v.begin();
int maxElement = *std::max_element(v.begin(), v.end());

int minElementIndex = std::min_element(v.begin(),v.end()) - v.begin();
int minElement = *std::min_element(v.begin(), v.end());

std::cout << "maxElementIndex:" << maxElementIndex << ", maxElement:" << maxElement << '\n';
std::cout << "minElementIndex:" << minElementIndex << ", minElement:" << minElement << '\n';
```

Ausgabe:

> maxElementIndex: 3, maxElement: 10
> minElementIndex: 1, minElement: 2

C ++ 11

Das Minimum- und Maximum-Element in einem Vektor kann gleichzeitig mit der Methode `std::minmax_element` , die auch im Header `<algorithm>` definiert ist:

```
std::vector<int> v = {5, 2, 8, 10, 9};
auto minmax = std::minmax_element(v.begin(), v.end());

std::cout << "minimum element: " << *minmax.first << '\n';
std::cout << "maximum element: " << *minmax.second << '\n';
```

Ausgabe:

> Mindestelement: 2
> Maximales Element: 10

Matrizen mit Vektoren

Vektoren können als 2D-Matrix verwendet werden, indem sie als Vektor von Vektoren definiert werden.

Eine Matrix mit 3 Zeilen und 4 Spalten, wobei jede Zelle als 0 initialisiert wird, kann folgendermaßen definiert werden:

```
std::vector<std::vector<int> > matrix(3, std::vector<int>(4));
```

C ++ 11

Die Syntax für die Initialisierung mit Initialisierungslisten oder auf andere Weise ist der eines normalen Vektors ähnlich.

```
  std::vector<std::vector<int>> matrix = { {0,1,2,3},
                                           {4,5,6,7},
                                           {8,9,10,11}
                                         };
```

Auf Werte in einem solchen Vektor kann ähnlich wie bei einem 2D-Array zugegriffen werden

```
int var = matrix[0][2];
```

Die Iteration über die gesamte Matrix ist der eines normalen Vektors ähnlich, jedoch mit einer zusätzlichen Dimension.

```
for(int i = 0; i < 3; ++i)
{
    for(int j = 0; j < 4; ++j)
    {
        std::cout << matrix[i][j] << std::endl;
    }
}
```

C ++ 11

```
for(auto& row: matrix)
{
    for(auto& col : row)
    {
        std::cout << col << std::endl;
    }
}
```

Ein Vektor von Vektoren ist eine bequeme Art, eine Matrix darzustellen, ist jedoch nicht die effizienteste: Einzelne Vektoren sind im Speicher verteilt und die Datenstruktur ist nicht für den Cache geeignet.

In einer richtigen Matrix muss die Länge jeder Zeile gleich sein (dies ist bei einem Vektor von Vektoren nicht der Fall). Die zusätzliche Flexibilität kann eine Fehlerquelle sein.

std :: vector online lesen: https://riptutorial.com/de/cplusplus/topic/511/std----vector

Kapitel 125: Stream-Manipulatoren

Einführung

Manipulatoren sind spezielle Hilfsfunktionen, mit deren Hilfe Eingabe- und Ausgabeströme mit dem `operator >>` oder `operator <<` gesteuert werden.

Sie können alle mit `#include <iomanip>` .

Bemerkungen

Manipulatoren können auf andere Weise verwendet werden. Zum Beispiel:

1. `os.width(n);` entspricht `os << std::setw(n);`
   `is.width(n);` entspricht `is >> std::setw(n);`

2. `os.precision(n);` entspricht `os << std::setprecision(n);`
   `is.precision(n);` entspricht `is >> std::setprecision(n);`

3. `os.setfill(c);` entspricht `os << std::setfill(c);`

4. `str >> std::setbase(base);` oder `str << std::setbase(base);` ist gleich

```
str.setf(base ==  8 ? std::ios_base::oct :
            base == 10 ? std::ios_base::dec :
                base == 16 ? std::ios_base::hex :
                     std::ios_base::fmtflags(0),
         std::ios_base::basefield);
```

5. `os.setf(std::ios_base::flag);` entspricht `os << std::flag;`
   `is.setf(std::ios_base::flag);` entspricht `is >> std::flag;`
   `os.unsetf(std::ios_base::flag);` entspricht `os << std::no ## flag;`
   `is.unsetf(std::ios_base::flag);` entspricht `is >> std::no ## flag;`
   (wobei ## - *Verkettungsoperator ist*)
   für die nächste `flag` s: **`boolalpha`** , **`showbase`** , **`showpoint`** , **`showpos`** , **`skipws`** , **`uppercase`** .

6. **`std::ios_base::basefield`** .
   Für `flag` s: **`dec`** , **`hex`** und **`oct`** :

- `os.setf(std::ios_base::flag, std::ios_base::basefield);` entspricht `os << std::flag;`
  `is.setf(std::ios_base::flag, std::ios_base::basefield);` entspricht `is >> std::flag;`
  (1)
- `str.unsetf(std::ios_base::flag, std::ios_base::basefield);` entspricht
  `str.setf(std::ios_base::fmtflags(0), std::ios_base::basefield);`
  (2)

7. **`std::ios_base::adjustfield`** .
   Für `flag` s: **`left`** , **`right`** und **`internal`** :

- `os.setf(std::ios_base::flag, std::ios_base::adjustfield);` entspricht `os << std::flag;`
  `is.setf(std::ios_base::flag, std::ios_base::adjustfield);` entspricht `is >> std::flag;`
  (1)
- `str.unsetf(std::ios_base::flag, std::ios_base::adjustfield);` entspricht
  `str.setf(std::ios_base::fmtflags(0), std::ios_base::adjustfield);`
  (2)

(1) Wenn das Flag des entsprechenden Felds zuvor durch `unsetf` .
(2) Wenn das `flag` gesetzt ist.

8. **`std::ios_base::floatfield`** .

- `os.setf(std::ios_base::flag, std::ios_base::floatfield);` entspricht `os << std::flag;`
  `is.setf(std::ios_base::flag, std::ios_base::floatfield);` entspricht `is >> std::flag;`
  für `flag` s: **`fixed`** und **`scientific`** .
- `os.setf(std::ios_base::fmtflags(0), std::ios_base::floatfield);` entspricht `os <<
  std::defaultfloat;`
  `is.setf(std::ios_base::fmtflags(0), std::ios_base::floatfield);` entspricht `is >>
  std::defaultfloat;`

9. `str.setf(std::ios_base::fmtflags(0), std::ios_base::flag);` entspricht
   `str.unsetf(std::ios_base::flag)`
   für `flag` s: **`basefield`** , **`adjustfield`** , **`floatfield`** .

10. `os.setf(mask)` entspricht `os << setiosflags(mask);`
    `is.setf(mask)` entspricht `is >> setiosflags(mask);`
    `os.unsetf(mask)` entspricht `os << resetiosflags(mask);`
    `is.unsetf(mask)` gleich `is >> resetiosflags(mask);`
    Für fast alle `mask` des Typs `std::ios_base::fmtflags` .

Examples

Stream-Manipulatoren

`std::boolalpha` und `std::noboolalpha` - zwischen textlicher und numerischer Darstellung von Booleschen wechseln.

```
std::cout << std::boolalpha << 1;
// Output: true

std::cout << std::noboolalpha << false;
// Output: 0

bool boolValue;
std::cin >> std::boolalpha >> boolValue;
std::cout << "Value \"" << std::boolalpha << boolValue
          << "\" was parsed as " << std::noboolalpha << boolValue;
// Input: true
// Output: Value "true" was parsed as 0
```

`std::showbase` und `std::noshowbase` - steuern, ob ein Präfix für die numerische Basis verwendet wird.

`std::dec` (dezimal), `std::hex` (hexadezimal) und `std::oct` (octal) - werden zum Ändern der Basis für Ganzzahlen verwendet.

```
#include <sstream>

std::cout << std::dec << 29 << ' - '
          << std::hex << 29 << ' - '
          << std::showbase << std::oct << 29 << ' - '
          << std::noshowbase << 29  '\n';
int number;
std::istringstream("3B") >> std::hex >> number;
std::cout << std::dec << 10;
// Output: 22 - 1D - 35 - 035
// 59
```

Standardwerte sind `std::ios_base::noshowbase` und `std::ios_base::dec` .

Wenn Sie mehr über `std::istringstream` überprüfen Sie den < sstream > -Header.

`std::uppercase` und `std::nouppercase` - steuern, ob Großbuchstaben in der Ausgabe von Fließkomma- und Hexadezimal-Integer verwendet werden. Habe keine Auswirkung auf Eingabeströme.

```
std::cout << std::hex << std::showbase
              << "0x2a with nouppercase: " << std::nouppercase << 0x2a << '\n'
              << "1e-10 with uppercase: " << std::uppercase << 1e-10 << '\n'
}
// Output: 0x2a with nouppercase: 0x2a
// 1e-10 with uppercase: 1E-10
```

Standard ist `std::nouppercase` .

`std::setw(n)` - ändert die Breite des nächsten Eingabe- / Ausgabefeldes auf genau `n` .

Die width-Eigenschaft `n` wird auf `0` wenn einige Funktionen aufgerufen werden (vollständige Liste ist hier).

```
std::cout << "no setw:" << 51 << '\n'
          << "setw(7): " << std::setw(7) << 51 << '\n'
          << "setw(7), more output: " << 13
          << std::setw(7) << std::setfill('*') << 67 << ' ' << 94 << '\n';

char* input = "Hello, world!";
char arr[10];
std::cin >> std::setw(6) >> arr;
std::cout << "Input from \"Hello, world!\" with setw(6) gave \"" << arr << "\"\n";

// Output: 51
// setw(7):      51
// setw(7), more output: 13*****67 94

// Input: Hello, world!
// Output: Input from "Hello, world!" with setw(6) gave "Hello"
```

Standard ist `std::setw(0)` .

`std::left` , `std::right` und `std::internal` - Ändern Sie die Standardposition der Füllzeichen, indem Sie `std::ios_base::adjustfield` auf `std::ios_base::left` , `std::ios_base::right` und `std::ios_base::internal` entsprechend. `std::left` und `std::right` gelten für jede Ausgabe, `std::internal` - für Ganzzahl-, Gleitkomma- und Geldausgabe. Habe keine Auswirkung auf Eingabeströme.

```
#include <locale>
...

std::cout.imbue(std::locale("en_US.utf8"));

std::cout << std::left << std::showbase << std::setfill('*')
          << "flt: " << std::setw(15) << -9.87  << '\n'
          << "hex: " << std::setw(15) << 41 << '\n'
          << "  $: " << std::setw(15) << std::put_money(367, false) << '\n'
          << "usd: " << std::setw(15) << std::put_money(367, true) << '\n'
          << "usd: " << std::setw(15)
          << std::setfill(' ') << std::put_money(367, false) << '\n';
// Output:
// flt: -9.87**********
// hex: 41*************
//   $: $3.67**********
// usd: USD *3.67******
// usd: $3.67

std::cout << std::internal << std::showbase << std::setfill('*')
```

```
          << "flt: " << std::setw(15) << -9.87  << '\n'
          << "hex: " << std::setw(15) << 41 << '\n'
          << "  $: " << std::setw(15) << std::put_money(367, false) << '\n'
          << "usd: " << std::setw(15) << std::put_money(367, true) << '\n'
          << "usd: " << std::setw(15)
          << std::setfill(' ') << std::put_money(367, true) << '\n';
// Output:
// flt: -**********9.87
// hex: *************41
//   $: $3.67**********
// usd: USD *******3.67
// usd: USD        3.67

std::cout << std::right << std::showbase << std::setfill('*')
          << "flt: " << std::setw(15) << -9.87  << '\n'
          << "hex: " << std::setw(15) << 41 << '\n'
          << "  $: " << std::setw(15) << std::put_money(367, false) << '\n'
          << "usd: " << std::setw(15) << std::put_money(367, true) << '\n'
          << "usd: " << std::setw(15)
          << std::setfill(' ') << std::put_money(367, true) << '\n';
// Output:
// flt: **********-9.87
// hex: *************41
//   $: **********$3.67
// usd: ******USD *3.67
// usd:       USD  3.67
```

Standard ist `std::left` .

`std::fixed` , `std::scientific` , `std::hexfloat` [C ++ 11] und `std::defaultfloat` [C ++ 11] - Formatierung für Gleitkomma-Eingabe / Ausgabe ändern.

`std::fixed` setzt `std::ios_base::floatfield` auf `std::ios_base::fixed` ,
`std::scientific` - bis `std::ios_base::scientific` ,
`std::hexfloat` - bis `std::ios_base::fixed | std::ios_base::scientific` und
`std::defaultfloat` - bis `std::ios_base::fmtflags(0)` .

`fmtflags`

```
#include <sstream>
...

std::cout << '\n'
          << "The number 0.07 in fixed:      " << std::fixed << 0.01 << '\n'
          << "The number 0.07 in scientific: " << std::scientific << 0.01 << '\n'
          << "The number 0.07 in hexfloat:   " << std::hexfloat << 0.01 << '\n'
          << "The number 0.07 in default:    " << std::defaultfloat << 0.01 << '\n';

double f;
std::istringstream is("0x1P-1022");
double f = std::strtod(is.str().c_str(), NULL);
std::cout << "Parsing 0x1P-1022 as hex gives " << f << '\n';

// Output:
// The number 0.01 in fixed:      0.070000
```

```
// The number 0.01 in scientific: 7.000000e-02
// The number 0.01 in hexfloat:   0x1.1eb851eb851ecp-4
// The number 0.01 in default:    0.07
// Parsing 0x1P-1022 as hex gives 2.22507e-308
```

Der Standardwert ist `std::ios_base::fmtflags(0)` .

Bei einigen Compilern liegt ein **Fehler** vor, der die Ursache hat

```
double f;
std::istringstream("0x1P-1022") >> std::hexfloat >> f;
std::cout << "Parsing 0x1P-1022 as hex gives " << f << '\n';
// Output: Parsing 0x1P-1022 as hex gives 0
```

`std::showpoint` und `std::noshowpoint` - steuern, ob in der Fließkommadarstellung immer ein Dezimalpunkt enthalten ist. Habe keine Auswirkung auf Eingabeströme.

```
std::cout << "7.0 with showpoint: " << std::showpoint << 7.0 << '\n'
          << "7.0 with noshowpoint: " << std::noshowpoint << 7.0 << '\n';
// Output: 1.0 with showpoint: 7.00000
// 1.0 with noshowpoint: 7
```

Standard ist `std::showpoint` .

`std::showpos` und `std::noshowpos` - Steuerung Anzeige der `+` Zeichen in *nicht-negativen* Ausgang. Habe keine Auswirkung auf Eingabeströme.

```
std::cout << "With showpos: " << std::showpos
          << 0 << ' ' << -2.718 << ' ' << 17 << '\n'
          << "Without showpos: " << std::noshowpos
          << 0 << ' ' << -2.718 << ' ' << 17 << '\n';
// Output: With showpos: +0 -2.718 +17
// Without showpos: 0 -2.718 17
```

Standardeinstellung wenn `std::noshowpos` .

`std::unitbuf` , `std::nounitbuf` - Kontrolle des Ausgabestroms nach jeder Operation. Habe keine Auswirkung auf den Eingabestrom. `std::unitbuf` führt zum Spülen.

`std::setbase(base)` - `std::setbase(base)` die numerische Basis des Streams fest.

`std::setbase(8)` entspricht der Einstellung von `std::ios_base::basefield` auf `std::ios_base::oct`
`std::setbase(16)` - bis `std::ios_base::hex` ,
`std::setbase(10)` - bis `std::ios_base::dec` .

Wenn `base` anders als `8` , `10` oder `16` ist, wird `std::ios_base::basefield` auf `std::ios_base::fmtflags(0)` . Es bedeutet dezimale Ausgabe und präfixabhängige Eingabe.

Standardmäßig ist `std::ios_base::basefield` `std::ios_base::dec` und standardmäßig `std::setbase(10)` .

`std::setprecision(n)` - ändert die Gleitkomma-Genauigkeit.

```
#include <cmath>
#include <limits>
...

typedef std::numeric_limits<long double> ld;
const long double pi = std::acos(-1.L);

std::cout << '\n'
          << "default precision (6):   pi: " << pi << '\n'
          << "                        10pi: " << 10 * pi << '\n'
          << "std::setprecision(4):  10pi: " << std::setprecision(4) << 10 * pi << '\n'
          << "                    10000pi: " << 10000 * pi << '\n'
          << "std::fixed:         10000pi: " << std::fixed << 10000 * pi << std::defaultfloat
<< '\n'
          << "std::setprecision(10):   pi: " << std::setprecision(10) << pi << '\n'
          << "max-1 radix precicion:   pi: " << std::setprecision(ld::digits - 1) << pi <<
'\n'
          << "max+1 radix precision:   pi: " << std::setprecision(ld::digits + 1) << pi <<
'\n'
          << "significant digits prec: pi: " << std::setprecision(ld::digits10) << pi << '\n';

// Output:
// default precision (6):   pi: 3.14159
//                         10pi: 31.4159
// std::setprecision(4):  10pi: 31.42
//                     10000pi: 3.142e+04
// std::fixed:         10000pi: 31415.9265
// std::setprecision(10):   pi: 3.141592654
// max-1 radix precicion:   pi:
3.1415926535897932385128089594061862044327426701784133911132812 5
// max+1 radix precision:   pi:
3.1415926535897932385128089594061862044327426701784133911132812 5
// significant digits prec: pi: 3.14159265358979324
```

Standard ist `std::setprecision(6)` .

`std::setiosflags(mask)` und `std::resetiosflags(mask)` - Setzen und Löschen von Flags, die in der `mask` vom Typ `std::ios_base::fmtflags` .

```
#include <sstream>
...

std::istringstream in("10 010 10 010 10 010");
int num1, num2;
```

```
in >> std::oct >> num1 >> num2;
std::cout << "Parsing \"10 010\" with std::oct gives:   " << num1 << ' ' << num2 << '\n';
// Output: Parsing "10 010" with std::oct gives:   8 8

in >> std::dec >> num1 >> num2;
std::cout << "Parsing \"10 010\" with std::dec gives:   " << num1 << ' ' << num2 << '\n';
// Output: Parsing "10 010" with std::oct gives:   10 10

in >> std::resetiosflags(std::ios_base::basefield) >> num1 >> num2;
std::cout << "Parsing \"10 010\" with autodetect gives: " << num1 << ' ' << num2 << '\n';
// Parsing "10 010" with autodetect gives: 10 8

std::cout << std::setiosflags(std::ios_base::hex |
                              std::ios_base::uppercase |
                              std::ios_base::showbase) << 42 << '\n';
// Output: OX2A
```

`std::skipws` und `std::noskipws` - steuern Sie das Überspringen des führenden Leerraums durch die formatierten Eingabefunktionen. Habe keine Auswirkung auf Ausgabeströme.

```
#include <sstream>
...

char c1, c2, c3;
std::istringstream("a b c") >> c1 >> c2 >> c3;
std::cout << "Default  behavior:  c1 = " << c1 << "  c2 = " << c2 << "  c3 = " << c3 << '\n';

std::istringstream("a b c") >> std::noskipws >> c1 >> c2 >> c3;
std::cout << "noskipws behavior:  c1 = " << c1 << "  c2 = " << c2 << "  c3 = " << c3 << '\n';
// Output: Default  behavior:  c1 = a  c2 = b  c3 = c
// noskipws behavior:  c1 = a  c2 =    c3 = b
```

Die Standardeinstellung ist `std::ios_base::skipws` .

`std::quoted(s[, delim[, escape]])` [C ++ 14] - fügt zitierte Zeichenfolgen mit eingebetteten Leerzeichen ein oder extrahiert sie.

`s` - die Zeichenkette, die eingefügt oder extrahiert werden soll.
`delim` - das Zeichen als Trennzeichen zu verwenden, `"` standardmäßig aktiviert .
`escape` - das Zeichen als Escape - Zeichen zu verwenden, `\` standardmäßig.

```
#include <sstream>
...

std::stringstream ss;
std::string in = "String with spaces, and embedded \"quotes\" too";
std::string out;

ss << std::quoted(in);
std::cout << "read in     [" << in << "]\n"
          << "stored as   [" << ss.str() << "]\n";
```

```
ss >> std::quoted(out);
std::cout << "written out [" << out << "]\n";
// Output:
// read in     [String with spaces, and embedded "quotes" too]
// stored as   ["String with spaces, and embedded \"quotes\" too"]
// written out [String with spaces, and embedded "quotes" too]
```

Weitere Informationen finden Sie unter dem Link oben.

Ausgabestrommanipulatoren

`std::ends` - fügt ein Nullzeichen `'\0'` in den Ausgabestrom ein. Formaler sieht diese Manipulator-Deklaration aus

```
template <class charT, class traits>
std::basic_ostream<charT, traits>& ends(std::basic_ostream<charT, traits>& os);
```

und dieser Manipulator platziert ein Zeichen durch Aufrufen von `os.put(charT())` bei Verwendung in einem Ausdruck
`os << std::ends;`

`std::endl` und `std::flush` beides spülen den Ausgabestrom ab `out` indem Sie `out.flush()` aufrufen. Dadurch wird sofort eine Ausgabe erzeugt. Aber `std::endl` fügt das Zeilenende `'\n'` vor dem Löschen ein.

```
std::cout << "First line." << std::endl << "Second line. " << std::flush
          << "Still second line.";
// Output: First line.
// Second line. Still second line.
```

`std::setfill(c)` - ändert das Füllzeichen in `c` . `std::setw` häufig mit `std::setw` .

```
std::cout << "\nDefault fill: " << std::setw(10) << 79 << '\n'
          << "setfill('#'): " << std::setfill('#')
          << std::setw(10) << 42 << '\n';
// Output:
// Default fill:         79
// setfill('#'): ########79
```

`std::put_money(mon[, intl])` [C ++ 11]. In einem Ausdruck `out << std::put_money(mon, intl)` konvertiert der monetäre Wert `mon` (vom Typ `long double` oder `std::basic_string`) in seine Zeichendarstellung, wie von der `std::money_put` Facette des aktuell verwendeten Gebietsschemas angegeben in `out` Verwenden Sie internationale Währungszeichenfolgen, wenn `intl true` , verwenden Sie ansonsten Währungssymbole.

```
long double money = 123.45;
// or std::string money = "123.45";

std::cout.imbue(std::locale("en_US.utf8"));
std::cout << std::showbase << "en_US: " << std::put_money(money)
          << " or " << std::put_money(money, true) << '\n';
// Output: en_US: $1.23 or USD  1.23

std::cout.imbue(std::locale("ru_RU.utf8"));
std::cout << "ru_RU: " << std::put_money(money)
          << " or " << std::put_money(money, true) << '\n';
// Output: ru_RU: 1.23 руб or 1.23 RUB

std::cout.imbue(std::locale("ja_JP.utf8"));
std::cout << "ja_JP: " << std::put_money(money)
          << " or " << std::put_money(money, true) << '\n';
// Output: ja_JP: ￥123 or JPY  123
```

`std::put_time(tmb, fmt)` [C ++ 11] - formatiert und gibt einen Datums- / Zeitwert gemäß dem angegebenen Format `fmt` an `std::tm` .

`tmb` - Zeiger auf die Kalenderzeitstruktur `const std::tm*` wie von `localtime()` oder `gmtime()` .
`fmt` - Zeiger auf einen nullterminierten String `const CharT*` , der das Format der Konvertierung angibt.

```
#include <ctime>
...

std::time_t t = std::time(nullptr);
std::tm tm = *std::localtime(&t);

std::cout.imbue(std::locale("ru_RU.utf8"));
std::cout << "\nru_RU: " << std::put_time(&tm, "%c %Z") << '\n';
// Possible output:
// ru_RU: Вт 04 июл 2017 15:08:35 UTC
```

Weitere Informationen finden Sie unter dem Link oben.

Eingangsstrom-Manipulatoren

`std::ws` verbraucht führende Leerzeichen im Eingabestrom. Es unterscheidet sich von `std::skipws` .

```
#include <sstream>
...

std::string str;
std::istringstream("  \v\n\r\t    Wow!There   is no whitespaces!") >> std::ws >> str;
std::cout << str;
// Output: Wow!There   is no whitespaces!
```

`std::get_money(mon[, intl])` [C ++ 11]. In einem Ausdruck `in >> std::get_money(mon, intl)` die Zeicheneingabe als monetärer Wert analysiert, wie von der `std::money_get` Facette des aktuell in `in` `std::money_get` Gebietsschemas angegeben, und der Wert wird in `mon` (`long double` `std::money_get` `long double` oder `std::basic_string` type). Der Manipulator erwartet die *erforderlichen* internationalen Währungszeichenfolgen, wenn `intl` `true` ist. Andernfalls werden *optionale* Währungssymbole erwartet.

```
#include <sstream>
#include <locale>
...

std::istringstream in("$1,234.56 2.22 USD  3.33");
long double v1, v2;
std::string v3;

in.imbue(std::locale("en_US.UTF-8"));
in >> std::get_money(v1) >> std::get_money(v2) >> std::get_money(v3, true);
if (in) {
    std::cout << std::quoted(in.str()) << " parsed as: "
              << v1 << ", " << v2 << ", " << v3 << '\n';
}
// Output:
// "$1,234.56 2.22 USD  3.33" parsed as: 123456, 222, 333
```

`std::get_time(tmb, fmt)` [C ++ 11] - analysiert einen Datums- / Zeitwert, der in `tmb` des angegebenen Formats `fmt` gespeichert ist.

`tmb` - gültiger Zeiger auf das Objekt `const std::tm*` , in dem das Ergebnis gespeichert wird.
`fmt` - Zeiger auf eine mit Null abgeschlossene Zeichenfolge `const CharT*` , die das Konvertierungsformat angibt.

```
#include <sstream>
#include <locale>
...

std::tm t = {};
std::istringstream ss("2011-Februar-18 23:12:34");

ss.imbue(std::locale("de_DE.utf-8"));
ss >> std::get_time(&t, "%Y-%b-%d %H:%M:%S");
if (ss.fail()) {
    std::cout << "Parse failed\n";
}
else {
    std::cout << std::put_time(&t, "%c") << '\n';
}
// Possible output:
// Sun Feb 18 23:12:34 2011
```

Weitere Informationen finden Sie unter dem Link oben.

Stream-Manipulatoren online lesen: https://riptutorial.com/de/cplusplus/topic/10699/stream-manipulatoren

Kapitel 126: Thread-Synchronisationsstrukturen

Einführung

Die Arbeit mit Threads erfordert möglicherweise einige Synchronisationstechniken, wenn die Threads interagieren. In diesem Thema finden Sie die verschiedenen Strukturen, die von der Standardbibliothek bereitgestellt werden, um diese Probleme zu lösen.

Examples

std :: shared_lock

Ein shared_lock kann zusammen mit einer eindeutigen Sperre verwendet werden, um mehrere Leser und exklusive Schreiber zuzulassen.

```
#include <unordered_map>
#include <mutex>
#include <shared_mutex>
#include <thread>
#include <string>
#include <iostream>

class PhoneBook {
    public:
        string getPhoneNo( const std::string & name )
        {
            shared_lock<shared_timed_mutex> r(_protect);
            auto it =  _phonebook.find( name );
            if ( it == _phonebook.end() )
                return (*it).second;
            return "";
        }
        void addPhoneNo ( const std::string & name, const std::string & phone )
        {
            unique_lock<shared_timed_mutex> w(_protect);
            _phonebook[name] = phone;
        }

        shared_timed_mutex _protect;
        unordered_map<string,string>  _phonebook;
    };
```

std :: call_once, std :: once_flag

`std::call_once` stellt die Ausführung einer Funktion durch konkurrierende Threads genau einmal sicher. Es wirft `std::system_error` aus, falls es seine Aufgabe nicht erfüllen kann.

Wird zusammen mit s `td::once_flag` .

```
#include <mutex>
#include <iostream>

std::once_flag flag;
void do_something(){
      std::call_once(flag, [](){std::cout << "Happens once" << std::endl;});

      std::cout << "Happens every time" << std::endl;
}
```

Objektverriegelung für einen effizienten Zugriff.

Oft möchten Sie das gesamte Objekt sperren, während Sie mehrere Vorgänge daran ausführen. Zum Beispiel, wenn Sie das Objekt mithilfe von *Iteratoren* untersuchen oder ändern müssen. Wenn Sie mehrere Elementfunktionen aufrufen müssen, ist es generell effizienter, das gesamte Objekt als einzelne Elementfunktionen zu sperren.

Zum Beispiel:

```
class text_buffer
{
    // for readability/maintainability
    using mutex_type = std::shared_timed_mutex;
    using reading_lock = std::shared_lock<mutex_type>;
    using updates_lock = std::unique_lock<mutex_type>;

public:
    // This returns a scoped lock that can be shared by multiple
    // readers at the same time while excluding any writers
    [[nodiscard]]
    reading_lock lock_for_reading() const { return reading_lock(mtx); }

    // This returns a scoped lock that is exclusing to one
    // writer preventing any readers
    [[nodiscard]]
    updates_lock lock_for_updates() { return updates_lock(mtx); }

    char* data() { return buf; }
    char const* data() const { return buf; }

    char* begin() { return buf; }
    char const* begin() const { return buf; }

    char* end() { return buf + sizeof(buf); }
    char const* end() const { return buf + sizeof(buf); }

    std::size_t size() const { return sizeof(buf); }

private:
    char buf[1024];
    mutable mutex_type mtx; // mutable allows const objects to be locked
};
```

Bei der Berechnung einer Prüfsumme wird das Objekt zum Lesen gesperrt, sodass andere Threads, die gleichzeitig aus dem Objekt lesen möchten, dies tun können.

```
std::size_t checksum(text_buffer const& buf)
{
    std::size_t sum = 0xA44944A4;

    // lock the object for reading
    auto lock = buf.lock_for_reading();

    for(auto c: buf)
        sum = (sum << 8) | (((unsigned char) ((sum & 0xFF000000) >> 24)) ^ c);

    return sum;
}
```

Durch das Löschen des Objekts werden seine internen Daten aktualisiert, sodass eine Ausschließungssperre erforderlich ist.

```
void clear(text_buffer& buf)
{
    auto lock = buf.lock_for_updates(); // exclusive lock
    std::fill(std::begin(buf), std::end(buf), '\0');
}
```

Wenn Sie mehr als eine Sperre erhalten, müssen Sie darauf achten, dass die Sperren für alle Threads immer in derselben Reihenfolge abgerufen werden.

```
void transfer(text_buffer const& input, text_buffer& output)
{
    auto lock1 = input.lock_for_reading();
    auto lock2 = output.lock_for_updates();

    std::copy(std::begin(input), std::end(input), std::begin(output));
}
```

Anmerkung: Dies geschieht am besten mit std :: deferred :: lock und dem Aufruf von std :: lock

std :: condition_variable_any, std :: cv_status

Eine Verallgemeinerung von `std::condition_variable` , `std::condition_variable_any` funktioniert mit jeder Art von BasicLockable-Struktur.

`std::cv_status` als Rückgabestatus für eine Bedingungsvariable hat zwei mögliche Rückgabecodes:

- std :: cv_status :: no_timeout: Es gab kein Timeout, die Zustandsvariable wurde benachrichtigt
- std :: cv_status :: no_timeout: Zeitlimit für Zustandsvariable

Thread-Synchronisationsstrukturen online lesen:
https://riptutorial.com/de/cplusplus/topic/9794/thread-synchronisationsstrukturen

Kapitel 127: Tools und Techniken zum Debuggen und Debuggen von C ++

Einführung

Das Debuggen wird viel Zeit von C ++ - Entwicklern aufgewendet. Dieses Thema soll bei dieser Aufgabe helfen und Techniken inspirieren. Erwarten Sie keine umfangreiche Liste von Problemen und Lösungen, die von den Tools oder einem Handbuch zu den genannten Tools behoben werden.

Bemerkungen

Dieses Thema ist noch nicht abgeschlossen. Beispiele zu folgenden Techniken / Tools wären nützlich:

- Erwähnen Sie weitere statische Analysewerkzeuge
- Werkzeuge für binäre Instrumente (wie UBSan, TSan, MSan, ESan ...)
- Härten (CFI ...)
- Fuzzing

Examples

Mein C ++ - Programm endet mit segfault-valgrind

Lassen Sie uns ein grundlegendes Fehlerprogramm haben:

```
#include <iostream>

void fail() {
    int *p1;
    int *p2(NULL);
    int *p3 = p1;
    if (p3) {
        std::cout << *p3 << std::endl;
    }
}

int main() {
    fail();
}
```

Erstellen Sie es (fügen Sie -g hinzu, um Debug-Informationen einzuschließen):

```
g++ -g -o main main.cpp
```

Lauf:

```
$ ./main
Segmentation fault (core dumped)
$
```

Lass uns mit valgrind debuggen:

```
$ valgrind ./main
==8515== Memcheck, a memory error detector
==8515== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==8515== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==8515== Command: ./main
==8515==
==8515== Conditional jump or move depends on uninitialised value(s)
==8515==    at 0x400813: fail() (main.cpp:7)
==8515==    by 0x40083F: main (main.cpp:13)
==8515==
==8515== Invalid read of size 4
==8515==    at 0x400819: fail() (main.cpp:8)
==8515==    by 0x40083F: main (main.cpp:13)
==8515==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
==8515==
==8515==
==8515== Process terminating with default action of signal 11 (SIGSEGV): dumping core
==8515==  Access not within mapped region at address 0x0
==8515==    at 0x400819: fail() (main.cpp:8)
==8515==    by 0x40083F: main (main.cpp:13)
==8515==  If you believe this happened as a result of a stack
==8515==  overflow in your program's main thread (unlikely but
==8515==  possible), you can try to increase the size of the
==8515==  main thread stack using the --main-stacksize= flag.
==8515==  The main thread stack size used in this run was 8388608.
==8515==
==8515== HEAP SUMMARY:
==8515==     in use at exit: 72,704 bytes in 1 blocks
==8515==   total heap usage: 1 allocs, 0 frees, 72,704 bytes allocated
==8515==
==8515== LEAK SUMMARY:
==8515==    definitely lost: 0 bytes in 0 blocks
==8515==    indirectly lost: 0 bytes in 0 blocks
==8515==      possibly lost: 0 bytes in 0 blocks
==8515==    still reachable: 72,704 bytes in 1 blocks
==8515==         suppressed: 0 bytes in 0 blocks
==8515== Rerun with --leak-check=full to see details of leaked memory
==8515==
==8515== For counts of detected and suppressed errors, rerun with: -v
==8515== Use --track-origins=yes to see where uninitialised values come from
==8515== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
$
```

Zuerst konzentrieren wir uns auf diesen Block:

```
==8515== Invalid read of size 4
==8515==    at 0x400819: fail() (main.cpp:8)
==8515==    by 0x40083F: main (main.cpp:13)
==8515==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

Die erste Zeile sagt uns, dass Segfault durch das Lesen von 4 Bytes verursacht wird. Die zweite und die dritte Leitung sind Anrufstapel. Das bedeutet, dass der ungültige Lesevorgang in der

`fail()` Funktion in Zeile 8 von main.cpp ausgeführt wird, die von main, Zeile 13 von main.cpp aufgerufen wird.

Wenn wir Zeile 8 von main.cpp betrachten, sehen wir

```
std::cout << *p3 << std::endl;
```

Aber wir prüfen zuerst den Zeiger. Was ist los? Lass uns den anderen Block überprüfen:

```
==8515== Conditional jump or move depends on uninitialised value(s)
==8515==    at 0x400813: fail() (main.cpp:7)
==8515==    by 0x40083F: main (main.cpp:13)
```

Es sagt uns, dass es in Zeile 7 eine unitialisierte Variable gibt, und wir lesen sie:

```
if (p3) {
```

Was zeigt uns auf die Zeile, wo wir p3 statt p2 überprüfen. Aber wie ist es möglich, dass p3 nicht initialisiert wird? Wir initialisieren es durch:

```
int *p3 = p1;
```

Valgrind rät uns mit `--track-origins=yes` , lass es uns tun:

```
valgrind --track-origins=yes ./main
```

Das Argument für Valgrind ist kurz nach Valgrind. Wenn wir es nach unserem Programm stellen, wird es an unser Programm weitergegeben.

Die Ausgabe ist fast gleich, es gibt nur einen Unterschied:

```
==8517== Conditional jump or move depends on uninitialised value(s)
==8517==    at 0x400813: fail() (main.cpp:7)
==8517==    by 0x40083F: main (main.cpp:13)
==8517==  Uninitialised value was created by a stack allocation
==8517==    at 0x4007F6: fail() (main.cpp:3)
```

Was uns sagt, dass der nicht initialisierte Wert, den wir in Zeile 7 verwendeten, in Zeile 3 erstellt wurde:

```
int *p1;
```

das führt uns zu unserem nicht initialisierten Zeiger.

Segfault-Analyse mit GDB

Verwenden wir den gleichen Code wie oben für dieses Beispiel.

```
#include <iostream>

void fail() {
    int *p1;
    int *p2(NULL);
    int *p3 = p1;
    if (p3) {
        std::cout << *p2 << std::endl;
    }
}

int main() {
    fail();
}
```

Zuerst können wir es kompilieren

```
g++ -g -o main main.cpp
```

Lass es mit gdb laufen

```
gdb ./main
```

Jetzt werden wir in gdb shell sein. Geben Sie run ein.

```
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/opencog/code-snippets/stackoverflow/a.out

Program received signal SIGSEGV, Segmentation fault.
0x0000000000400850 in fail () at debugging_with_gdb.cc:11
11            std::cout << *p2 << std::endl;
```

Wir sehen, dass der Segmentierungsfehler in Zeile 11 auftritt. Daher wird in dieser Zeile nur der Zeiger p2 verwendet. Lässt den Inhalt des Druckvorgangs prüfen.

```
(gdb) print p2
$1 = (int *) 0x0
```

Nun sehen wir, dass p2 auf 0x0 initialisiert wurde, was für NULL steht. In dieser Zeile wissen wir, dass wir versuchen, einen NULL-Zeiger zu dereferenzieren. Also gehen wir los und reparieren es.

Code reinigen

Das Debuggen beginnt mit dem Verstehen des Codes, den Sie debuggen möchten.

Falscher Code:

```
int main() {
    int value;
    std::vector<int> vectorToSort;
```

```
    vectorToSort.push_back(42); vectorToSort.push_back(13);
    for (int i = 52; i; i = i - 1)
    {
    vectorToSort.push_back(i *2);
    }
    /// Optimized for sorting small vectors
    if (vectorToSort.size() == 1);
    else
        {
        if (vectorToSort.size() <= 2)
            std::sort(vectorToSort.begin(), std::end(vectorToSort));
        }
    for (value : vectorToSort) std::cout << value << ' ';
return 0; }
```

Besserer Code:

```
std::vector<int> createSemiRandomData() {
    std::vector<int> data;
    data.push_back(42);
    data.push_back(13);
    for (int i = 52; i; --i)
        vectorToSort.push_back(i *2);
    return data;
}

/// Optimized for sorting small vectors
void sortVector(std::vector &v) {
    if (vectorToSort.size() == 1)
        return;
    if (vectorToSort.size() > 2)
        return;

    std::sort(vectorToSort.begin(), vectorToSort.end());
}

void printVector(const std::vector<int> &v) {
    for (auto i : v)
        std::cout << i << ' ';
}

int main() {
    auto vectorToSort = createSemiRandomData();
    sortVector(std::ref(vectorToSort));
    printVector(vectorToSort);

    return 0;
 }
```

Unabhängig von den von Ihnen bevorzugten und verwendeten Codierungsstilen hilft Ihnen ein einheitlicher Codierungsstil (und Formatierungsstil), den Code besser zu verstehen.

Wenn Sie den Code oben betrachten, können Sie einige Verbesserungen feststellen, um die Lesbarkeit und die Fehlersuche zu verbessern:

Die Verwendung separater Funktionen für separate Aktionen

Durch die Verwendung separater Funktionen können Sie einige Funktionen im Debugger überspringen, wenn Sie nicht an den Details interessiert sind. In diesem speziellen Fall sind Sie möglicherweise nicht an der Erstellung oder dem Ausdruck der Daten interessiert und möchten nur in die Sortierung einsteigen.

Ein weiterer Vorteil ist, dass Sie beim Durchlaufen des Codes weniger Code lesen und speichern müssen. Sie müssen jetzt nur 3 Zeilen Code in `main()` lesen, um es zu verstehen, anstatt die ganze Funktion zu verwenden.

Der dritte Vorteil ist, dass Sie einfach weniger Code zum Anschauen haben, was einem geschulten Auge hilft, diesen Fehler innerhalb von Sekunden zu erkennen.

Verwenden konsistenter Formatierungen / Konstruktionen

Durch die Verwendung konsistenter Formatierungen und Konstruktionen wird Unordnung aus dem Code entfernt, sodass Sie sich leichter auf den Code statt auf Text konzentrieren können. Es wurden viele Diskussionen über den "richtigen" Formatierungsstil geführt. Unabhängig von diesem Stil wird durch die Verwendung eines einzigen konsistenten Stils im Code die Vertrautheit verbessert und es wird einfacher, sich auf den Code zu konzentrieren.

Da Formatierungscode eine zeitaufwändige Aufgabe ist, wird empfohlen, hierfür ein spezielles Werkzeug zu verwenden. Die meisten IDEs haben zumindest eine gewisse Unterstützung dafür und können konsistentere Formatierungen als Menschen vornehmen.

Möglicherweise ist der Stil nicht auf Leerzeichen und Zeilenumbrüche beschränkt, da der freie Stil und die Member-Funktionen nicht mehr gemischt werden, um den Anfang und das Ende des Containers zu erhalten. (`v.begin()` vs `std::end(v)`).

Machen Sie auf die wichtigen Teile Ihres Codes aufmerksam.

Unabhängig davon, welchen Stil Sie wählen, enthält der obige Code einige Markierungen, die Ihnen einen Hinweis auf das geben könnten, was wichtig sein könnte:

- Ein Kommentar mit `optimized` Angaben weist auf einige ausgefallene Techniken hin
- Einige frühe Rückgaben in `sortVector()` zeigen an, dass wir etwas Besonderes tun
- Die `std::ref()` zeigt an, dass mit `sortVector()` etwas `sortVector()`

Fazit

Mit sauberem Code können Sie den Code besser verstehen und die Zeit für das Debuggen reduzieren. Im zweiten Beispiel kann ein Code-Reviewer den Fehler sogar auf den ersten Blick erkennen, während der Fehler in den Details des ersten Fehlers versteckt ist. (PS: Der Fehler ist im Vergleich mit `2`)

Statische Analyse

Statische Analyse ist die Technik, bei der der Code auf Muster überprüft wird, die mit bekannten Fehlern verknüpft sind. Die Verwendung dieser Technik ist weniger zeitaufwändig als eine Codeüberprüfung. Die Überprüfung ist jedoch nur auf die im Tool programmierten beschränkt.

Bei Prüfungen kann das falsche Semikolon hinter der if-Anweisung (`if (var);`) bis zu fortgeschrittenen Graph-Algorithmen stehen, die bestimmen, ob eine Variable nicht initialisiert wird.

Compiler-Warnungen

Das Aktivieren der statischen Analyse ist einfach. Die einfachste Version ist bereits in Ihrem Compiler integriert:

- `clang++ -Wall -Weverything -Werror ...`
- `g++ -Wall -Weverything -Werror ...`
- `cl.exe /W4 /WX ...`

Wenn Sie diese Optionen aktivieren, werden Sie feststellen, dass jeder Compiler Fehler findet, die andere nicht tun, und dass Sie Fehler in Bezug auf Techniken erhalten, die in einem bestimmten Kontext gültig oder gültig sind. `while (staticAtomicBool);` kann auch dann akzeptiert werden, wenn `while (localBool);` ist nicht

Anders als bei der Codeüberprüfung kämpfen Sie mit einem Tool, das Ihren Code versteht, viele nützliche Fehler anzeigt und manchmal mit Ihnen nicht einverstanden ist. In diesem letzten Fall müssen Sie die Warnung möglicherweise lokal unterdrücken.

Da die obigen Optionen alle Warnungen aktivieren, werden möglicherweise Warnungen aktiviert, die Sie nicht möchten. (Warum sollte Ihr Code C ++ 98-kompatibel sein?) Wenn dies der Fall ist, können Sie diese Warnung einfach deaktivieren:

- `clang++ -Wall -Weverything -Werror -Wno-errortoaccept ...`
- `g++ -Wall -Weverything -Werror -Wno-errortoaccept ...`
- `cl.exe /W4 /WX /wd<no of warning>...`

Wo Compiler-Warnungen Sie während der Entwicklung unterstützen, verlangsamen sie die Kompilierung erheblich. Aus diesem Grund möchten Sie sie möglicherweise nicht standardmäßig aktivieren. Entweder führen Sie sie standardmäßig aus, oder Sie ermöglichen eine fortlaufende Integration der teureren (oder aller) Prüfungen.

Externe Werkzeuge

Wenn Sie sich für eine fortlaufende Integration entscheiden, ist die Verwendung anderer Tools nicht so groß. Ein Tool wie clang-tidy enthält eine Liste von Überprüfungen, die ein breites Spektrum von Problemen abdeckt, einige Beispiele:

- Aktuelle Fehler
  - Verhinderung des Schneidens
  - Assays mit Nebenwirkungen
- Lesbarkeitsprüfungen

    - Irreführende Einrückung
    - Überprüfen Sie die Benennung der IDs
- Modernisierungsprüfungen
    - Verwenden Sie make_unique ()
    - Verwenden Sie nullptr
- Leistungsprüfungen
    - Finden Sie nicht benötigte Exemplare
    - Finden Sie ineffiziente Algorithmusaufrufe

Die Liste ist möglicherweise nicht so umfangreich, da Clang bereits viele Compiler-Warnungen enthält, Sie werden jedoch einer qualitativ hochwertigen Codebasis einen Schritt näher kommen.

Andere Werkzeuge

Andere Tools mit ähnlichem Zweck existieren, wie:

- das visual studio static analyzer als externes Werkzeug
- clazy , ein Clang-Compiler-Plugin zur Überprüfung von Qt-Code

Fazit

Es gibt viele statische Analysewerkzeuge für C ++, die beide als externe Werkzeuge in den Compiler integriert sind. Das Ausprobieren kostet nicht viel Zeit für einfache Setups und sie werden Fehler finden, die Sie bei der Code-Überprüfung vermissen könnten.

Safe-Stack (Stack-Korruption)

Stapelverfälschungen sind ärgerliche Fehler. Da der Stapel beschädigt ist, kann der Debugger Ihnen oft keine gute Übersicht darüber geben, wo Sie sich befinden und wie Sie dorthin gekommen sind.

Hier kommt Safe-Stack ins Spiel. Anstatt einen einzelnen Stack für Ihre Threads zu verwenden, werden zwei verwendet: ein sicherer Stack und ein gefährlicher Stack. Der sichere Stapel funktioniert genauso wie zuvor, nur dass einige Teile in den gefährlichen Stapel verschoben werden.

Welche Teile des Stapels werden verschoben?

Jedes Teil, das möglicherweise den Stapel beschädigt, wird aus dem sicheren Stapel verschoben. Sobald eine Variable auf dem Stack als Referenz übergeben wird oder die Adresse dieser Variablen abgerufen wird, entscheidet der Compiler, diese auf dem zweiten Stack statt auf dem sicheren Stack zuzuordnen.

Daher können bei jeder Operation, die Sie mit diesen Zeigern ausführen, Änderungen am Speicher (basierend auf diesen Zeigern / Referenzen) nur den Speicher im zweiten Stapel beeinflussen. Da man nie einen Zeiger erhält, der sich in der Nähe des sicheren Stapels befindet,

kann der Stapel den Stapel nicht beschädigen, und der Debugger kann trotzdem alle Funktionen auf dem Stapel lesen, um eine nette Spur zu erhalten.

Wofür wird es eigentlich verwendet?

Der sichere Stack wurde nicht entwickelt, um Ihnen ein besseres Debugging-Erlebnis zu bieten, ist jedoch ein schöner Nebeneffekt für böse Fehler. Ihr ursprünglicher Zweck ist Teil des CPI-Projekts (Code-Pointer Integrity) , in dem versucht wird, das Überschreiben der Rücksprungadressen zu verhindern, um die Code-Injektion zu verhindern. Mit anderen Worten, sie versuchen zu verhindern, dass ein Hackercode ausgeführt wird.

Aus diesem Grunde hat die Funktion wurde auf Chrom aktiviert und wurde berichtet , einen <1% CPU - Overhead haben.

Wie kann ich es aktivieren?

Im Moment ist die Option nur im Clang-Compiler verfügbar, wo `-fsanitize=safe-stack` an den Compiler übergeben werden kann. Ein Vorschlag gemacht wurde die gleiche Funktion in GCC zu implementieren.

Fazit

Stapelverfälschungen können einfacher zu debuggen werden, wenn der sichere Stapel aktiviert ist. Aufgrund eines geringen Leistungsaufwands können Sie sogar standardmäßig in Ihrer Build-Konfiguration aktiviert werden.

Tools und Techniken zum Debuggen und Debuggen von C ++ online lesen: https://riptutorial.com/de/cplusplus/topic/9814/tools-und-techniken-zum-debuggen-und-debuggen-von-c-plusplus

Kapitel 128: Typ Inferenz

Einführung

In diesem Thema wird die Typinferenzierung beschrieben, die den von C ++ 11 verfügbaren Schlüsselwort `auto` type umfasst.

Bemerkungen

Es ist normalerweise besser, `const` , `&` und `constexpr` zu deklarieren, wenn Sie `auto` wenn unerwünschte Verhaltensweisen wie Kopieren oder Mutationen `constexpr` . Diese zusätzlichen Hinweise stellen sicher, dass der Compiler keine anderen Inferenzformen generiert. Es ist auch nicht ratsam, die Funktion `auto` überschreiben, und sollte nur verwendet werden, wenn die tatsächliche Deklaration sehr lang ist, insbesondere bei STL-Vorlagen.

Examples

Datentyp: Auto

Dieses Beispiel zeigt die grundlegenden Typinterferenzen, die der Compiler ausführen kann.

```
auto a = 1;         //    a = int
auto b = 2u;        //    b = unsigned int
auto c = &a;        //    c = int*
const auto  d = c; //    d = const int*
const auto& e = b; //    e = const unsigned int&

auto x = a + b     //    x = int, #compiler warning unsigned and signed

auto v = std::vector<int>;    //    v = std::vector<int>
```

Das auto-Schlüsselwort führt jedoch nicht immer die erwartete Typinferenz ohne zusätzliche Hinweise für `&` oder `const` oder `constexpr`

```
//    y = unsigned int,
//    note that y does not infer as const unsigned int&
//    The compiler would have generated a copy instead of a reference value to e or b
auto y = e;
```

Lambda Auto

Der Datentyp "auto" ist eine bequeme Möglichkeit für Programmierer, Lambda-Funktionen zu deklarieren. Es hilft, indem die Anzahl der Textprogrammierer verkürzt wird, die zur Deklaration eines Funktionszeigers eingegeben werden müssen.

```
auto DoThis = [](int a, int b) { return a + b; };
```

```
//    Do this is of type (int)(*DoThis)(int, int)
//    else we would have to write this long
int(*pDoThis)(int, int)= [](int a, int b) { return a + b; };

auto c = Dothis(1, 2);    //    c = int
auto d = pDothis(1, 2);   //    d = int

//    using 'auto' shortens the definition for lambda functions
```

Wenn der Rückgabetyp von Lambda-Funktionen nicht definiert ist, wird er standardmäßig automatisch aus den Rückgabewerttypen abgeleitet.

Diese 3 sind im Grunde dasselbe

```
[](int a, int b) -> int  { return a + b; };
[](int a, int b) -> auto { return a + b; };
[](int a, int b) { return a + b; };
```

Schleifen und Auto

Dieses Beispiel zeigt, wie mit auto die Typdeklaration für for-Schleifen verkürzt werden kann

```
std::map<int, std::string> Map;
for (auto pair : Map)             //    pair = std::pair<int, std::string>
for (const auto pair : Map)       //    pair = const std::pair<int, std::string>
for (const auto& pair : Map)      //    pair = const std::pair<int, std::string>&
for (auto i = 0; i < 1000; ++i)         //    i = int
for (auto i = 0; i < Map.size(); ++i)   //    Note that i = int and not size_t
for (auto i = Map.size(); i > 0; --i)   //    i = size_t
```

Typ Inferenz online lesen: https://riptutorial.com/de/cplusplus/topic/8233/typ-inferenz

Kapitel 129: Typ löschen

Einführung

Typlöschung ist ein Satz von Techniken zum Erstellen eines Typs, der eine einheitliche Schnittstelle zu verschiedenen zugrunde liegenden Typen bereitstellen kann, während die zugrunde liegenden Typinformationen vor dem Client ausgeblendet werden. `std::function<R(A...)>` , die aufrufbare Objekte verschiedener Typen enthalten kann, ist vielleicht das bekannteste Beispiel für das Löschen von Typen in C ++.

Examples

Grundmechanismus

Mit der Typenlöschung können Sie den Typ eines Objekts mithilfe von Code vor Code ausblenden, auch wenn es nicht von einer allgemeinen Basisklasse abgeleitet ist. Dadurch wird eine Brücke zwischen den Welten des statischen Polymorphismus (Templates; am Ort der Verwendung muss der genaue Typ zum Zeitpunkt des Kompilierens bekannt sein, muss aber nicht als konform mit einer Schnittstelle definiert werden) und dynamischem Polymorphismus (Vererbung und virtuelle Funktionen; am Ort der Verwendung muss der genaue Typ nicht zur Kompilierzeit bekannt sein, sondern muss bei der Definition als konform mit einer Schnittstelle deklariert werden).

Der folgende Code zeigt den grundlegenden Mechanismus zum Löschen von Typen.

```
#include <ostream>

class Printable
{
public:
  template <typename T>
  Printable(T value) : pValue(new Value<T>(value)) {}
  ~Printable() { delete pValue; }
  void print(std::ostream &os) const { pValue->print(os); }

private:
  Printable(Printable const &)        /* in C++1x: =delete */; // not implemented
  void operator = (Printable const &) /* in C++1x: =delete */; // not implemented
  struct ValueBase
  {
      virtual ~ValueBase() = default;
      virtual void print(std::ostream &) const = 0;
  };
  template <typename T>
  struct Value : ValueBase
  {
      Value(T const &t) : v(t) {}
      virtual void print(std::ostream &os) const { os << v; }
      T v;
  };
```

```
  ValueBase *pValue;
};
```

Auf der Benutzungssite muss nur die obige Definition sichtbar sein, genau wie bei Basisklassen mit virtuellen Funktionen. Zum Beispiel:

```
#include <iostream>

void print_value(Printable const &p)
{
    p.print(std::cout);
}
```

Beachten Sie, dass dies *keine* Vorlage ist, sondern eine normale Funktion, die nur in einer Header-Datei deklariert werden muss und in einer Implementierungsdatei definiert werden kann (im Gegensatz zu Vorlagen, deren Definition am Verwendungsort sichtbar sein muss).

Bei der Definition des konkreten Typs muss über `Printable` nichts bekannt sein, es muss lediglich eine Schnittstelle wie bei Vorlagen verwendet werden:

```
struct MyType { int i; };
ostream& operator << (ostream &os, MyType const &mc)
{
  return os << "MyType {" << mc.i << "}";
}
```

Wir können nun ein Objekt dieser Klasse an die oben definierte Funktion übergeben:

```
MyType foo = { 42 };
print_value(foo);
```

Löschen auf einen normalen Typ mit manueller vtable

C ++ basiert auf einem so genannten Regular-Typ (oder zumindest Pseudo-Regular).

Ein regulärer Typ ist ein Typ, der durch Kopieren oder Verschieben konstruiert und zugewiesen und zugewiesen werden kann, der zerstört werden kann und mit dem verglichen werden kann. Es kann auch ohne Argumente konstruiert werden. Schließlich werden auch einige andere Operationen unterstützt, die in verschiedenen `std` und Containern sehr nützlich sind.

Dies ist das Stammpapier , aber in C ++ 11 möchte die Unterstützung für `std::hash` hinzugefügt werden.

Ich werde hier den manuellen vtable-Ansatz verwenden, um das Löschen von Texten vorzunehmen.

```
using dtor_unique_ptr = std::unique_ptr<void, void(*)(void*)>;
template<class T, class...Args>
dtor_unique_ptr make_dtor_unique_ptr( Args&&... args ) {
  return {new T(std::forward<Args>(args)...), [](void* self){ delete static_cast<T*>(self);
}};
```

```
}
struct regular_vtable {
  void(*copy_assign)(void* dest, void const* src); // T&=(T const&)
  void(*move_assign)(void* dest, void* src); // T&=(T&&)
  bool(*equals)(void const* lhs, void const* rhs); // T const&==T const&
  bool(*order)(void const* lhs, void const* rhs); // std::less<T>{}(T const&, T const&)
  std::size_t(*hash)(void const* self); // std::hash<T>{}(T const&)
  std::type_info const&(*type)(); // typeid(T)
  dtor_unique_ptr(*clone)(void const* self); // T(T const&)
};

template<class T>
regular_vtable make_regular_vtable() noexcept {
  return {
    [](void* dest, void const* src){ *static_cast<T*>(dest) = *static_cast<T const*>(src); },
    [](void* dest, void* src){ *static_cast<T*>(dest) = std::move(*static_cast<T*>(src)); },
    [](void const* lhs, void const* rhs){ return *static_cast<T const*>(lhs) == *static_cast<T
const*>(rhs); },
    [](void const* lhs, void const* rhs) { return std::less<T>{}(*static_cast<T
const*>(lhs),*static_cast<T const*>(rhs)); },
    [](void const* self){ return std::hash<T>{}(*static_cast<T const*>(self)); },
    []()->decltype(auto){ return typeid(T); },
    [](void const* self){ return make_dtor_unique_ptr<T>(*static_cast<T const*>(self)); }
  };
}
template<class T>
regular_vtable const* get_regular_vtable() noexcept {
  static const regular_vtable vtable=make_regular_vtable<T>();
  return &vtable;
}

struct regular_type {
  using self=regular_type;
  regular_vtable const* vtable = 0;
  dtor_unique_ptr ptr{nullptr, [](void*){}};

  bool empty() const { return !vtable; }

  template<class T, class...Args>
  void emplace( Args&&... args ) {
    ptr = make_dtor_unique_ptr<T>(std::forward<Args>(args)...);
    if (ptr)
      vtable = get_regular_vtable<T>();
    else
      vtable = nullptr;
  }
  friend bool operator==(regular_type const& lhs, regular_type const& rhs) {
    if (lhs.vtable != rhs.vtable) return false;
    return lhs.vtable->equals( lhs.ptr.get(), rhs.ptr.get() );
  }
  bool before(regular_type const& rhs) const {
    auto const& lhs = *this;
    if (!lhs.vtable || !rhs.vtable)
      return std::less<regular_vtable const*>{}(lhs.vtable,rhs.vtable);
    if (lhs.vtable != rhs.vtable)
      return lhs.vtable->type().before(rhs.vtable->type());
    return lhs.vtable->order( lhs.ptr.get(), rhs.ptr.get() );
  }
  // technically friend bool operator< that calls before is also required

  std::type_info const* type() const {
```

```
    if (!vtable) return nullptr;
    return &vtable->type();
  }
  regular_type(regular_type&& o):
    vtable(o.vtable),
    ptr(std::move(o.ptr))
  {
    o.vtable = nullptr;
  }
  friend void swap(regular_type& lhs, regular_type& rhs){
    std::swap(lhs.ptr, rhs.ptr);
    std::swap(lhs.vtable, rhs.vtable);
  }
  regular_type& operator=(regular_type&& o) {
    if (o.vtable == vtable) {
      vtable->move_assign(ptr.get(), o.ptr.get());
      return *this;
    }
    auto tmp = std::move(o);
    swap(*this, tmp);
    return *this;
  }
  regular_type(regular_type const& o):
    vtable(o.vtable),
    ptr(o.vtable?o.vtable->clone(o.ptr.get()):dtor_unique_ptr{nullptr, [](void*){}})
  {
    if (!ptr && vtable) vtable = nullptr;
  }
  regular_type& operator=(regular_type const& o) {
    if (o.vtable == vtable) {
      vtable->copy_assign(ptr.get(), o.ptr.get());
      return *this;
    }
    auto tmp = o;
    swap(*this, tmp);
    return *this;
  }
  std::size_t hash() const {
    if (!vtable) return 0;
    return vtable->hash(ptr.get());
  }
  template<class T,
    std::enable_if_t< !std::is_same<std::decay_t<T>, regular_type>{}, int>* =nullptr
  >
  regular_type(T&& t) {
    emplace<std::decay_t<T>>(std::forward<T>(t));
  }
};
namespace std {
  template<>
  struct hash<regular_type> {
    std::size_t operator()( regular_type const& r )const {
      return r.hash();
    }
  };
  template<>
  struct less<regular_type> {
    bool operator()( regular_type const& lhs, regular_type const& rhs ) const {
      return lhs.before(rhs);
    }
  };
```

```
}
```

Live-Beispiel .

Ein solcher regulärer Typ kann als Schlüssel für eine `std::map` oder eine `std::unordered_map` , die *reguläre* `std::unordered_map` für einen Schlüssel akzeptiert, wie z.

```
std::map<regular_type, std::any>
```

Im Grunde wäre es eine Karte von einem anderen regulären zu allem, was kopierbar ist.

Anders als `any` führt mein `regular_type` keine Optimierung für kleine Objekte durch und unterstützt auch nicht die `regular_type` der Originaldaten. Den ursprünglichen Typ zurückzubekommen, ist nicht schwer.

Kleine Objekt Optimierung erfordert , dass wir speichern einen ausgerichteten Speicherpuffer innerhalb der `regular_type` und sorgfältig die deleter des zwicken `ptr` nur das Objekt zu zerstören und sie nicht löschen.

Ich würde bei `make_dtor_unique_ptr` und es lehren, wie man manchmal die Daten in einem Puffer speichert, und dann im Heap, wenn kein Speicherplatz im Puffer vorhanden ist. Das kann ausreichen.

Eine Nur-Bewegung-Funktion "std ::"

`std::function` löscht bis auf wenige Operationen. Voraussetzung ist, dass der gespeicherte Wert kopierbar ist.

Dies führt zu Problemen in einigen Kontexten, z. B. bei Lambdas, die eindeutige Ptrs speichern. Wenn Sie die `std::function` in einem Kontext verwenden, in dem das Kopieren keine Rolle spielt, z. B. in einem Thread-Pool, in dem Sie Aufgaben an Threads versenden, kann diese Anforderung Overhead verursachen.

Insbesondere ist `std::packaged_task<Sig>` ein aufrufbares Objekt, das nur zum Verschieben verwendet wird. Sie können eine `std::packaged_task<R(Args...)>` in einer `std::packaged_task<void(Args...)>` , aber dies ist eine ziemlich schwere und obskure Methode, um nur eine Bewegung zu erstellen aufrufbare Typlöschungsklasse.

Also die `task` . Dies zeigt, wie Sie einen einfachen `std::function` schreiben können. Ich weggelassen , um die Copy - Konstruktor (die einen bedeuten würde das Hinzufügen `clone` - Methode `details::task_pimpl<...>` als auch).

```
template<class Sig>
struct task;

// putting it in a namespace allows us to specialize it nicely for void return value:
namespace details {
  template<class R, class...Args>
  struct task_pimpl {
    virtual R invoke(Args&&...args) const = 0;
```

```
    virtual ~task_pimpl() {};
    virtual const std::type_info& target_type() const = 0;
  };

  // store an F.  invoke(Args&&...) calls the f
  template<class F, class R, class...Args>
  struct task_pimpl_impl:task_pimpl<R,Args...> {
    F f;
    template<class Fin>
    task_pimpl_impl( Fin&& fin ):f(std::forward<Fin>(fin)) {}
    virtual R invoke(Args&&...args) const final override {
      return f(std::forward<Args>(args)...);
    }
    virtual const std::type_info& target_type() const final override {
      return typeid(F);
    }
  };

  // the void version discards the return value of f:
  template<class F, class...Args>
  struct task_pimpl_impl<F,void,Args...>:task_pimpl<void,Args...> {
    F f;
    template<class Fin>
    task_pimpl_impl( Fin&& fin ):f(std::forward<Fin>(fin)) {}
    virtual void invoke(Args&&...args) const final override {
      f(std::forward<Args>(args)...);
    }
    virtual const std::type_info& target_type() const final override {
      return typeid(F);
    }
  };
};

template<class R, class...Args>
struct task<R(Args...)> {
  // semi-regular:
  task()=default;
  task(task&&)=default;
  // no copy

private:
  // aliases to make some SFINAE code below less ugly:
  template<class F>
  using call_r = std::result_of_t<F const&(Args...)>;
  template<class F>
  using is_task = std::is_same<std::decay_t<F>, task>;
public:
  // can be constructed from a callable F
  template<class F,
    // that can be invoked with Args... and converted-to-R:
    class= decltype( (R)(std::declval<call_r<F>>()) ),
    // and is not this same type:
    std::enable_if_t<!is_task<F>{}, int>* = nullptr
  >
  task(F&& f):
    m_pImpl( make_pimpl(std::forward<F>(f)) )
  {}

  // the meat: the call operator
  R operator()(Args... args)const {
        return m_pImpl->invoke( std::forward<Args>(args)... );
```

```
  }
  explicit operator bool() const {
    return (bool)m_pImpl;
  }
  void swap( task& o ) {
    std::swap( m_pImpl, o.m_pImpl );
  }
  template<class F>
  void assign( F&& f ) {
    m_pImpl = make_pimpl(std::forward<F>(f));
  }
  // Part of the std::function interface:
  const std::type_info& target_type() const {
    if (!*this) return typeid(void);
    return m_pImpl->target_type();
  }
  template< class T >
  T* target() {
    return target_impl<T>();
  }
  template< class T >
  const T* target() const {
    return target_impl<T>();
  }
  // compare with nullptr    :
  friend bool operator==( std::nullptr_t, task const& self ) { return !self; }
  friend bool operator==( task const& self, std::nullptr_t ) { return !self; }
  friend bool operator!=( std::nullptr_t, task const& self ) { return !!self; }
  friend bool operator!=( task const& self, std::nullptr_t ) { return !!self; }
private:
  template<class T>
  using pimpl_t = details::task_pimpl_impl<T, R, Args...>;

  template<class F>
  static auto make_pimpl( F&& f ) {
    using dF=std::decay_t<F>;
    using pImpl_t = pimpl_t<dF>;
    return std::make_unique<pImpl_t>(std::forward<F>(f));
  }
  std::unique_ptr<details::task_pimpl<R,Args...>> m_pImpl;

  template< class T >
  T* target_impl() const {
    return dynamic_cast<pimpl_t<T>*>(m_pImpl.get());
  }
};
```

Um diese Bibliothek würdig zu machen, möchten Sie eine kleine Pufferoptimierung hinzufügen, sodass nicht alle aufrufbaren Objekte auf dem Heap gespeichert werden.

Für das Hinzufügen von SBO sind eine nicht standardmäßige `task(task&&)` , ein Teil der Klasse `std::aligned_storage_t` , ein `m_pImpl` `unique_ptr` mit einem Deleter `m_pImpl` , der auf die Nur-Zerstörungsfunktion gesetzt werden kann (und nicht den Speicher an den Heap `emplace_move_to( void* ) = 0`) sowie ein `emplace_move_to( void* ) = 0` in `task_pimpl` .

Live-Beispiel für den obigen Code (ohne SBO).

Löschen in einen zusammenhängenden Puffer von T

Nicht alle Typen löschen virtuelle Vererbung, Zuweisung, Platzierung neuer oder sogar Funktionszeiger.

Was das Löschen von Typen bewirkt, ist das Beschreiben einer (Gruppe von) Verhalten (en) und die Verwendung eines beliebigen Typs, der dieses Verhalten unterstützt, und das Zusammenfassen. Alle Informationen, die nicht in dieser Gruppe von Verhaltensweisen enthalten sind, werden "vergessen" oder "gelöscht".

Ein `array_view` nimmt seinen eingehenden Bereich oder Containertyp und löscht alles, außer dass es ein zusammenhängender Puffer von `T` .

```
// helper traits for SFINAE:
template<class T>
using data_t = decltype( std::declval<T>().data() );

template<class Src, class T>
using compatible_data = std::integral_constant<bool, std::is_same< data_t<Src>, T* >{} ||
std::is_same< data_t<Src>, std::remove_const_t<T>* >{}>;

template<class T>
struct array_view {
  // the core of the class:
  T* b=nullptr;
  T* e=nullptr;
  T* begin() const { return b; }
  T* end() const { return e; }

  // provide the expected methods of a good contiguous range:
  T* data() const { return begin(); }
  bool empty() const { return begin()==end(); }
  std::size_t size() const { return end()-begin(); }

  T& operator[](std::size_t i)const{ return begin()[i]; }
  T& front()const{ return *begin(); }
  T& back()const{ return *(end()-1); }

  // useful helpers that let you generate other ranges from this one
  // quickly and safely:
  array_view without_front( std::size_t i=1 ) const {
    i = (std::min)(i, size());
    return {begin()+i, end()};
  }
  array_view without_back( std::size_t i=1 ) const {
    i = (std::min)(i, size());
    return {begin(), end()-i};
  }

  // array_view is plain old data, so default copy:
  array_view(array_view const&)=default;
  // generates a null, empty range:
  array_view()=default;

  // final constructor:
  array_view(T* s, T* f):b(s),e(f) {}
  // start and length is useful in my experience:
  array_view(T* s, std::size_t length):array_view(s, s+length) {}

  // SFINAE constructor that takes any .data() supporting container
```

```
  // or other range in one fell swoop:
  template<class Src,
    std::enable_if_t< compatible_data<std::remove_reference_t<Src>&, T >{}, int>* =nullptr,
    std::enable_if_t< !std::is_same<std::decay_t<Src>, array_view >{}, int>* =nullptr
  >
  array_view( Src&& src ):
    array_view( src.data(), src.size() )
  {}

  // array constructor:
  template<std::size_t N>
  array_view( T(&arr)[N] ):array_view(arr, N) {}

  // initializer list, allowing {} based:
  template<class U,
    std::enable_if_t< std::is_same<const U, T>{}, int>* =nullptr
  >
  array_view( std::initializer_list<U> il ):array_view(il.begin(), il.end()) {}
};
```

Ein `array_view` nimmt jeden Container an, der `.data()` , der einen Zeiger auf `T` und eine `.size()` -Methode oder ein Array `.size()` , und löscht diesen bis zu einem Bereich mit wahlfreiem Zugriff über zusammenhängende `T` .

Es kann einen `std::vector<T>` , einen `std::string<T>` ein `std::array<T, N>` a `T[37]` , eine Initialisierungsliste (einschließlich `{}` basierter) oder etwas anderes enthalten Sie machen das, was es unterstützt (über `T* x.data()` und `size_t x.size()`).

In diesem Fall bedeuten die Daten, die wir aus dem zu löschenden Objekt extrahieren können, zusammen mit dem Status "View", der sich nicht im Besitz des Besitzers befindet, dass wir keinen Speicher zuordnen müssen und keine benutzerdefinierten typabhängigen Funktionen schreiben müssen.

Live-Beispiel .

Eine Verbesserung wäre die Verwendung von Nicht-Mitglieder- `data` und einer Nicht-Mitglieder-`size` in einem ADL-fähigen Kontext.

Typlöschung mit std :: any löschen

In diesem Beispiel werden C ++ 14 und `boost::any` . In C ++ 17 können Sie stattdessen in `std::any` tauschen.

Die folgende Syntax lautet:

```
const auto print =
  make_any_method<void(std::ostream&)>([](auto&& p, std::ostream& t){ t << p << "\n"; });

super_any<decltype(print)> a = 7;

(a->*print)(std::cout);
```

das ist fast optimal.

Dieses Beispiel basiert auf der Arbeit von @dyp und @cpplearner sowie auf meinem eigenen.

Zuerst verwenden wir ein Tag, um Typen zu übergeben:

```
template<class T>struct tag_t{constexpr tag_t(){};};
template<class T>constexpr tag_t<T> tag{};
```

Diese Merkmalsklasse erhält die Signatur mit einer `any_method` gespeichert:

Dies erstellt einen Funktionszeigertyp und eine Factory für die Funktionszeiger, wenn eine `any_method` :

```
template<class any_method>
using any_sig_from_method = typename any_method::signature;

template<class any_method, class Sig=any_sig_from_method<any_method>>
struct any_method_function;

template<class any_method, class R, class...Args>
struct any_method_function<any_method, R(Args...)>
{
  template<class T>
  using decorate = std::conditional_t< any_method::is_const, T const, T >;

  using any = decorate<boost::any>;

  using type = R(*)(any&, any_method const*, Args&&...);
  template<class T>
  type operator()( tag_t<T> )const{
    return +[](any& self, any_method const* method, Args&&...args) {
      return (*method)( boost::any_cast<decorate<T>&>(self), decltype(args)(args)... );
    };
  }
};
```

`any_method_function::type` ist der Typ eines Funktionszeigers, der neben der Instanz gespeichert wird. `any_method_function::operator()` nimmt ein `tag_t<T>` und schreibt eine benutzerdefinierte Instanz des `any_method_function::type` , die annimmt, dass `any&` ein `T` .

Wir möchten in der Lage sein, mehr als eine Methode gleichzeitig zu löschen. Wir bündeln sie also in einem Tupel und schreiben einen Helper-Wrapper, um das Tupel pro Typ in statischen Speicher zu speichern und einen Zeiger auf sie aufzubewahren.

```
template<class...any_methods>
using any_method_tuple = std::tuple< typename any_method_function<any_methods>::type... >;

template<class...any_methods, class T>
any_method_tuple<any_methods...> make_vtable( tag_t<T> ) {
  return std::make_tuple(
    any_method_function<any_methods>{}(tag<T>)...
  );
}

template<class...methods>
```

```
struct any_methods {
private:
  any_method_tuple<methods...> const* vtable = 0;
  template<class T>
  static any_method_tuple<methods...> const* get_vtable( tag_t<T> ) {
    static const auto table = make_vtable<methods...>(tag<T>);
    return &table;
  }
public:
  any_methods() = default;
  template<class T>
  any_methods( tag_t<T> ): vtable(get_vtable(tag<T>)) {}
  any_methods& operator=(any_methods const&)=default;
  template<class T>
  void change_type( tag_t<T> ={} ) { vtable = get_vtable(tag<T>); }

  template<class any_method>
  auto get_invoker( tag_t<any_method> ={} ) const {
    return std::get<typename any_method_function<any_method>::type>( *vtable );
  }
};
```

Wir könnten dies auf Fälle spezialisieren, in denen die vtable klein ist (z. B. 1 Element), und direkte Zeiger verwenden, die in diesen Fällen in der Klasse gespeichert sind, um die Effizienz zu verbessern.

Jetzt starten wir die `super_any` . Ich benutze `super_any_t` , um die Deklaration von `super_any` etwas zu erleichtern.

```
template<class...methods>
struct super_any_t;
```

Dadurch werden die Methoden durchsucht, die Super Any für SFINAE und bessere Fehlermeldungen unterstützt:

```
template<class super_any, class method>
struct super_method_applies_helper : std::false_type {};

template<class M0, class...Methods, class method>
struct super_method_applies_helper<super_any_t<M0, Methods...>, method> :
    std::integral_constant<bool, std::is_same<M0, method>{}  ||
super_method_applies_helper<super_any_t<Methods...>, method>{}>
{};

template<class...methods, class method>
auto super_method_test( super_any_t<methods...> const&, tag_t<method> )
{
  return std::integral_constant<bool, super_method_applies_helper< super_any_t<methods...>,
method >{} && method::is_const >{};
}
template<class...methods, class method>
auto super_method_test( super_any_t<methods...>&, tag_t<method> )
{
  return std::integral_constant<bool, super_method_applies_helper< super_any_t<methods...>,
method >{} >{};
}
```

```
template<class super_any, class method>
struct super_method_applies:
    decltype( super_method_test( std::declval<super_any>(), tag<method> ) )
{};
```

Als Nächstes erstellen wir den Typ `any_method` . Eine `any_method` ist ein Pseudo-Methodenzeiger. Wir schaffen es global und `const` ly mit Syntax wie:

```
const auto print=make_any_method( [](auto&&self, auto&&os){ os << self; } );
```

oder in C ++ 17:

```
const any_method print=[](auto&&self, auto&&os){ os << self; };
```

Beachten Sie, dass die Verwendung eines Nicht-Lambdas die Sache haarig machen kann, da wir den Typ für einen Suchschritt verwenden. Dies kann behoben werden, würde aber dieses Beispiel länger machen, als es bereits ist. Initialisieren Sie daher immer eine beliebige Methode aus einem Lambda oder aus einem auf einem Lambda parametrisierten Typ.

```
template<class Sig, bool const_method, class F>
struct any_method {
  using signature=Sig;
  enum{is_const=const_method};
private:
  F f;
public:

  template<class Any,
    // SFINAE testing that one of the Anys's matches this type:
    std::enable_if_t< super_method_applies< Any&&, any_method >{}, int>* =nullptr
  >
  friend auto operator->*( Any&& self, any_method const& m ) {
    // we don't use the value of the any_method, because each any_method has
    // a unique type (!) and we check that one of the auto*'s in the super_any
    // already has a pointer to us.  We then dispatch to the corresponding
    // any_method_data...

    return [&self, invoke = self.get_invoker(tag<any_method>), m](auto&&...args)-
>decltype(auto)
    {
      return invoke( decltype(self)(self), &m, decltype(args)(args)... );
    };
  }
  any_method( F fin ):f(std::move(fin)) {}

  template<class...Args>
  decltype(auto) operator()(Args&&...args)const {
    return f(std::forward<Args>(args)...);
  }
};
```

Eine Factory-Methode, die in C ++ 17 nicht benötigt wird, glaube ich:

```
template<class Sig, bool is_const=false, class F>
any_method<Sig, is_const, std::decay_t<F>>
```

```
make_any_method( F&& f ) {
  return {std::forward<F>(f)};
}
```

Dies ist der Augmented `any` . Es ist sowohl eine `any` , und es trägt um ein Bündel von Typ-Löschfunktionszeiger, die sich ändern , wenn das enthaltene `any` tut:

```
template<class... methods>
struct super_any_t:boost::any, any_methods<methods...> {
  using vtable=any_methods<methods...>;
public:
  template<class T,
    std::enable_if_t< !std::is_base_of<super_any_t, std::decay_t<T>>{}, int> =0
  >
  super_any_t( T&& t ):
    boost::any( std::forward<T>(t) )
  {
    using dT=std::decay_t<T>;
    this->change_type( tag<dT> );
  }

  boost::any& as_any()&{return *this;}
  boost::any&& as_any()&&{return std::move(*this);}
  boost::any const& as_any()const&{return *this;}
  super_any_t()=default;
  super_any_t(super_any_t&& o):
    boost::any( std::move( o.as_any() ) ),
    vtable(o)
  {}
  super_any_t(super_any_t const& o):
    boost::any( o.as_any() ),
    vtable(o)
  {}
  template<class S,
    std::enable_if_t< std::is_same<std::decay_t<S>, super_any_t>{}, int> =0
  >
  super_any_t( S&& o ):
    boost::any( std::forward<S>(o).as_any() ),
    vtable(o)
  {}
  super_any_t& operator=(super_any_t&&)=default;
  super_any_t& operator=(super_any_t const&)=default;

  template<class T,
    std::enable_if_t< !std::is_same<std::decay_t<T>, super_any_t>{}, int>* =nullptr
  >
  super_any_t& operator=( T&& t ) {
    ((boost::any&)*this) = std::forward<T>(t);
    using dT=std::decay_t<T>;
    this->change_type( tag<dT> );
    return *this;
  }
};
```

Weil wir den Laden `any_method` s als `const` Objekte, macht dies zu einer machen `super_any` ein bisschen einfacher:

```
template<class...Ts>
```

```
using super_any = super_any_t< std::remove_cv_t<Ts>... >;
```

Testcode:

```
const auto print = make_any_method<void(std::ostream&)>([](auto&& p, std::ostream& t){ t << p
<< "\n"; });
const auto wprint = make_any_method<void(std::wostream&)>([](auto&& p, std::wostream& os ){ os
<< p << L"\n"; });

int main()
{
  super_any<decltype(print), decltype(wprint)> a = 7;
  super_any<decltype(print), decltype(wprint)> a2 = 7;

  (a->*print)(std::cout);
  (a->*wprint)(std::wcout);
}
```

Live-Beispiel .

Ursprünglich hier in einer SO-Frage & Antwort gepostet (und die oben genannten Personen halfen bei der Implementierung).

Typ löschen online lesen: https://riptutorial.com/de/cplusplus/topic/2872/typ-loschen

Kapitel 130: Typedef- und Typ-Aliase

Einführung

Die `typedef` und (seit C ++ 11) `using` von Schlüsselwörtern verwendet werden kann , um einen neuen Namen einer bestehenden Art zu geben.

Syntax

- typedef *Typspezifizierer-seq- Init-Deklaratorliste* ;
- *attributspezifizierer-seq* typedef *deklarationsbezeichner- seq init-deklaratorliste* ; // seit C ++ 11
- Verwendung *Identifikator attribute-Spezifizierer-Seq (opt) = type-ID;* // seit C ++ 11

Examples

Grundlegende Typedef-Syntax

Eine `typedef` Deklaration hat dieselbe Syntax wie eine Variablen- oder Funktionsdeklaration, enthält jedoch das Wort `typedef` . Das Vorhandensein von `typedef` bewirkt, dass die Deklaration einen Typ anstelle einer Variablen oder Funktion deklariert.

```
int T;         // T has type int
typedef int T; // T is an alias for int

int A[100];         // A has type "array of 100 ints"
typedef int A[100]; // A is an alias for the type "array of 100 ints"
```

Nachdem ein Typalias definiert wurde, kann er austauschbar mit dem ursprünglichen Namen des Typs verwendet werden.

```
typedef int A[100];
// S is a struct containing an array of 100 ints
struct S {
    A data;
};
```

`typedef` erstellt niemals einen eigenen Typ. Es gibt nur eine andere Möglichkeit, auf einen vorhandenen Typ zu verweisen.

```
struct S {
    int f(int);
};
typedef int I;
// ok: defines int S::f(int)
I S::f(I x) { return x; }
```

Komplexere Anwendungen von Typedef

Die Regel, dass `typedef` Deklarationen dieselbe Syntax wie gewöhnliche Variablen haben, und Funktionsdeklarationen können zum Lesen und Schreiben komplexer Deklarationen verwendet werden.

```
void (*f)(int);         // f has type "pointer to function of int returning void"
typedef void (*f)(int); // f is an alias for "pointer to function of int returning void"
```

Dies ist besonders nützlich für Konstrukte mit einer verwirrenden Syntax, z. B. Zeiger auf nicht statische Member.

```
void (Foo::*pmf)(int);         // pmf has type "pointer to member function of Foo taking int
                               // and returning void"
typedef void (Foo::*pmf)(int); // pmf is an alias for "pointer to member function of Foo
                               // taking int and returning void"
```

Die Syntax der folgenden Funktionsdeklarationen ist selbst für erfahrene Programmierer schwer zu merken:

```
void (Foo::*Foo::f(const char*))(int);
int (&g())[100];
```

`typedef` kann verwendet werden, um das Lesen und Schreiben zu erleichtern:

```
typedef void (Foo::pmf)(int);  // pmf is a pointer to member function type
pmf Foo::f(const char*);       // f is a member function of Foo

typedef int (&ra)[100];        // ra means "reference to array of 100 ints"
ra g();                        // g returns reference to array of 100 ints
```

Mehrere Typen mit typedef deklarieren

Das Schlüsselwort `typedef` ist ein `typedef` , es gilt also für jeden Deklarator. Daher bezieht sich jeder deklarierte Name auf den Typ, den dieser Name ohne `typedef` .

```
int *x, (*p)();         // x has type int*, and p has type int(*)()
typedef int *x, (*p)(); // x is an alias for int*, while p is an alias for int(*)()
```

Alias-Deklaration mit "using"

C ++ 11

Die Syntax der `using` ist sehr einfach: Der zu definierende Name wird auf der linken Seite und die Definition auf der rechten Seite angezeigt. Sie müssen nicht scannen, um zu sehen, wo der Name ist.

```
using I = int;
using A = int[100];             // array of 100 ints
```

```
using FP = void(*)(int);         // pointer to function of int returning void
using MP = void (Foo::*)(int);   // pointer to member function of Foo of int returning void
```

Das Erstellen eines `typedef` mit `using` hat genau dieselbe Wirkung wie das Erstellen eines `typedef` mit `typedef` . Es ist einfach eine alternative Syntax, um dasselbe zu erreichen.

Im Gegensatz zu `typedef` kann die `using` als Vorlage erfolgen. Eine „Vorlage typedef“ erstellt mit `using` wird eine genannt Alias - Vorlage .

Typedef- und Typ-Aliase online lesen: https://riptutorial.com/de/cplusplus/topic/9328/typedef--und-typ-aliase

Kapitel 131: Überladung des Bedieners

Einführung

In C ++ können Operatoren wie `+` und `->` für benutzerdefinierte Typen definiert werden. Beispielsweise definiert der Header `<string>` einen `+` -Operator zum Verketten von Zeichenfolgen. Dazu wird eine *Operatorfunktion* mit dem `operator` Schlüsselwort definiert .

Bemerkungen

Die Operatoren für integrierte Typen können nicht geändert werden. Operatoren können nur für benutzerdefinierte Typen überladen werden. Das heißt, mindestens einer der Operanden muss von einem benutzerdefinierten Typ sein.

Die folgenden Operatoren *können nicht* überladen werden:

- Der Mitgliederzugriff oder "Punkt" -Operator `.`
- Der Zeiger auf den Mitgliederzugriffsoperator `.*`
- Der Operator für die Bereichsauflösung `::`
- Der ternäre bedingte Operator `?:`
- `dynamic_cast` , `static_cast` , `reinterpret_cast` , `const_cast` , `typeid` , `sizeof` , `alignof` und `noexcept`
- Die Vorverarbeitungs-Direktiven `#` und `##` , die ausgeführt werden, bevor Typinformationen verfügbar sind.

Es gibt einige Operatoren, die Sie **nicht** (99,98% der Zeit) überlasten sollten:

- `&&` und `||` (bevorzugen Sie stattdessen die implizite Konvertierung in `bool`)
- `,`
- Die Adresse des Operators (unary `&`)

Warum? Sie überladen Operatoren, die ein anderer Programmierer möglicherweise niemals erwartet, was zu einem anderen Verhalten als erwartet führt.

Zum Beispiel die benutzerdefinierten `&&` und `||` Überlastungen dieser Operatoren ihre Kurzauswertung verlieren und ihre speziellen Eigenschaften Sequenzierung (C ++ 17) verlieren , gilt die Sequenzierung Problem auch `,` Betreiber Überlastung.

Examples

Rechenzeichen

Sie können alle grundlegenden arithmetischen Operatoren überladen:

- `+` und `+=`
- `-` und `-=`

- `*` und `*=`
- `/` und `/=`
- `&` und `&=`
- `|` und `|=`
- `^` und `^=`
- `>>` und `>>=`
- `<<` und `<<=`

Das Überladen für alle Bediener ist das gleiche. *Scrollen Sie zur Erklärung nach unten*

Überladung außerhalb der `class` / `struct` :

```
//operator+ should be implemented in terms of operator+=
T operator+(T lhs, const T& rhs)
{
    lhs += rhs;
    return lhs;
}

T& operator+=(T& lhs, const T& rhs)
{
    //Perform addition
    return lhs;
}
```

Überladung innerhalb von `class` / `struct` :

```
//operator+ should be implemented in terms of operator+=
T operator+(const T& rhs)
{
    *this += rhs;
    return *this;
}

T& operator+=(const T& rhs)
{
    //Perform addition
    return *this;
}
```

Hinweis: `operator+` sollte nicht konstanten Wert zurück, als eine Referenz Rückkehr nicht Sinn machen würde (es ein *neues* Objekt zurückgibt) , noch würde eine Rückkehr `const` Wert (Sie sollten in der Regel nicht wieder durch `const`). Das erste Argument wird als Wert übergeben, warum? weil

1. Sie können das ursprüngliche Objekt nicht ändern (`Object foobar = foo + bar;` sollte `foo` nicht ändern, es wäre nicht sinnvoll)
2. Sie können es nicht als `const` , da Sie das Objekt modifizieren müssen (weil `operator+` als `operator+=` implementiert ist, wodurch das Objekt geändert wird).

Das Übergeben von `const&` wäre eine Option, aber dann müssen Sie eine temporäre Kopie des übergebenen Objekts erstellen. Durch die Übergabe durch value erledigt der Compiler dies für

Sie.

`operator+=` gibt einen Verweis auf sich selbst zurück, da es dann möglich ist, sie zu verketten (verwenden Sie jedoch nicht dieselbe Variable, da dies aufgrund von Sequenzpunkten undefiniertes Verhalten wäre).

Das erste Argument ist eine Referenz (wir möchten sie ändern), aber nicht `const` , weil Sie sie dann nicht ändern können. Das zweite Argument sollte nicht geändert werden und wird daher aus Leistungsgründen von `const&` (das Übergeben von const-Referenz ist schneller als nach Wert).

Unäre Operatoren

Sie können die beiden unären Operatoren überladen:

- `++foo` und `foo++`
- `--foo` und `foo--`

Das Überladen ist für beide Typen (`++` und `--`) gleich. *Scrollen Sie zur Erklärung nach unten*

Überladung außerhalb der `class` / `struct` :

```
//Prefix operator ++foo
T& operator++(T& lhs)
{
    //Perform addition
    return lhs;
}

//Postfix operator foo++ (int argument is used to separate pre- and postfix)
//Should be implemented in terms of ++foo (prefix operator)
T operator++(T& lhs, int)
{
    T t(lhs);
    ++lhs;
    return t;
}
```

Überladung innerhalb von `class` / `struct` :

```
//Prefix operator ++foo
T& operator++()
{
    //Perform addition
    return *this;
}

//Postfix operator foo++ (int argument is used to separate pre- and postfix)
//Should be implemented in terms of ++foo (prefix operator)
T operator++(int)
{
    T t(*this);
    ++(*this);
    return t;
}
```

Hinweis: Der Präfixoperator gibt einen Verweis auf sich selbst zurück, sodass Sie die Vorgänge darauf fortsetzen können. Das erste Argument ist eine Referenz, da der Präfixoperator das Objekt ändert. Dies ist auch der Grund, warum es nicht `const` (sonst könnten Sie es nicht ändern).

Der Postfix-Operator gibt als Wert einen temporären Wert (den vorherigen Wert) zurück. Daher kann es sich nicht um einen Verweis handeln, da es sich um einen temporären Verweis handelt, der am Ende der Funktion ein Wert für die Verwendung von Darbietungen ist, da die temporäre Variable ausgeht Geltungsbereich). Es kann auch nicht `const` , da Sie es direkt ändern können sollten.

Das erste Argument ist eine nicht `const` Referenz auf das "aufrufende" Objekt, denn wenn es `const` wäre, könnten Sie es nicht ändern, und wenn es keine Referenz wäre, würden Sie den ursprünglichen Wert nicht ändern.

Es ist wegen des Kopierens in Postfix - Operator benötigt Überlastungen , dass es besser ist , es sich zur Gewohnheit zu machen Präfix ++ zu verwenden anstelle von Postfix ++ in `for` Schleifen. Aus der `for` Schleife-Perspektive sind sie normalerweise funktional gleichwertig, aber es kann einen geringfügigen Leistungsvorteil bei der Verwendung von Präfix ++ geben, insbesondere bei "fetten" Klassen mit vielen zu kopierenden Elementen. Beispiel für die Verwendung von Präfix ++ in einer for-Schleife:

```
for (list<string>::const_iterator it = tokens.begin();
     it != tokens.end();
     ++it) { // Don't use it++
    ...
}
```

Vergleichsoperatoren

Sie können alle Vergleichsoperatoren überladen:

- `==` und `!=`
- `>` und `<`
- `>=` und `<=`

Es wird empfohlen, alle diese Operatoren zu überladen, indem Sie nur zwei Operatoren (`==` und `<`) implementieren und diese dann verwenden, um den Rest zu definieren. *Scrollen Sie zur Erklärung nach unten*

Überladung außerhalb der `class` / `struct` :

```
//Only implement those 2
bool operator==(const T& lhs, const T& rhs) { /* Compare */ }
bool operator<(const T& lhs, const T& rhs) { /* Compare */ }

//Now you can define the rest
bool operator!=(const T& lhs, const T& rhs) { return !(lhs == rhs); }
bool operator>(const T& lhs, const T& rhs) { return rhs < lhs; }
bool operator<=(const T& lhs, const T& rhs) { return !(lhs > rhs); }
bool operator>=(const T& lhs, const T& rhs) { return !(lhs < rhs); }
```

Überladung innerhalb von `class` / `struct` :

```
//Note that the functions are const, because if they are not const, you wouldn't be able
//to call them if the object is const

//Only implement those 2
bool operator==(const T& rhs) const { /* Compare */ }
bool operator<(const T& rhs) const { /* Compare */ }

//Now you can define the rest
bool operator!=(const T& rhs) const { return !(*this == rhs); }
bool operator>(const T& rhs) const { return rhs < *this; }
bool operator<=(const T& rhs) const { return !(*this > rhs); }
bool operator>=(const T& rhs) const { return !(*this < rhs); }
```

Die Operatoren geben offensichtlich ein `bool` und geben für die entsprechende Operation `true` oder `false` an.

Alle Operatoren verwenden ihre Argumente durch `const&` , da die Operatoren nur Vergleichen durchführen und daher die Objekte nicht ändern dürfen. Das Übergeben von `&` (Referenz) ist schneller als mit dem Wert. Um sicherzustellen, dass die Operatoren es nicht ändern, handelt es sich um eine `const` Referenz.

Beachten Sie, dass die Operatoren in der `class` / `struct` als `const` definiert sind. Der Grund dafür ist, dass ohne die Funktionen von `const` Vergleich von `const` Objekten nicht möglich ist, da der Compiler nicht weiß, dass die Operatoren nichts ändern.

Konvertierungsoperatoren

Sie können Typoperatoren überladen, sodass Ihr Typ implizit in den angegebenen Typ konvertiert werden kann.

Der Konvertierungsoperator **muss** in einer `class` / `struct` :

```
operator T() const { /* return something */ }
```

Hinweis: Der Operator ist `const` , damit `const` Objekte konvertiert werden können.

Beispiel:

```
struct Text
{
    std::string text;

    // Now Text can be implicitly converted into a const char*
    /*explicit*/ operator const char*() const { return text.data(); }
    // ^^^^^^^^
    // to disable implicit conversion
};

Text t;
t.text = "Hello world!";
```

```
//Ok
const char* copyoftext = t;
```

Array-Indexoperator

Sie können sogar den Array-Indexoperator `[]` überladen.

Sie sollten **immer** (99,98% der Zeit) implementieren 2 Versionen, eine `const` und eine nicht- `const` Version, denn wenn das Objekt `const` , es nicht in der Lage sein sollte , das Objekt zurückgegeben von ändern `[]` .

Die Argumente werden von `const&` anstatt von value übergeben, da die Übergabe durch Verweis schneller ist als durch value und `const` sodass der Operator den Index nicht aus Versehen ändert.

Die Operatoren werden als Referenz zurückgegeben, da Sie das Objekt `[]` return modifizieren können, dh:

```
std::vector<int> v{ 1 };
v[0] = 2; //Changes value of 1 to 2
          //wouldn't be possible if not returned by reference
```

Sie können **nur** innerhalb einer `class` / `struct` überladen:

```
//I is the index type, normally an int
T& operator[](const I& index)
{
    //Do something
    //return something
}

//I is the index type, normally an int
const T& operator[](const I& index) const
{
    //Do something
    //return something
}
```

Mehrere Subskriptionsoperatoren `[][]...` können über Proxy-Objekte erreicht werden. Das folgende Beispiel einer einfachen Matrixmatrix mit Zeilenmaxiven veranschaulicht dies:

```
template<class T>
class matrix {
    // class enabling [][] overload to access matrix elements
    template <class C>
    class proxy_row_vector {
        using reference = decltype(std::declval<C>()[0]);
        using const_reference = decltype(std::declval<C const>()[0]);
    public:
        proxy_row_vector(C& _vec, std::size_t _r_ind, std::size_t _cols)
            : vec(_vec), row_index(_r_ind), cols(_cols) {}
        const_reference operator[](std::size_t _col_index) const {
```

```
            return vec[row_index*cols + _col_index];
        }
        reference operator[](std::size_t _col_index) {
            return vec[row_index*cols + _col_index];
        }
    private:
        C& vec;
        std::size_t row_index; // row index to access
        std::size_t cols; // number of columns in matrix
    };

    using const_proxy = proxy_row_vector<const std::vector<T>>;
    using proxy = proxy_row_vector<std::vector<T>>;
public:
    matrix() : mtx(), rows(0), cols(0) {}
    matrix(std::size_t _rows, std::size_t _cols)
        : mtx(_rows*_cols), rows(_rows), cols(_cols) {}

    // call operator[] followed by another [] call to access matrix elements
    const_proxy operator[](std::size_t _row_index) const {
        return const_proxy(mtx, _row_index, cols);
    }

    proxy operator[](std::size_t _row_index) {
        return proxy(mtx, _row_index, cols);
    }
private:
    std::vector<T> mtx;
    std::size_t rows;
    std::size_t cols;
};
```

Funktionsaufruf-Operator

Sie können den Funktionsaufrufoperator `()` überladen:

Das Überladen muss innerhalb einer `class` / `struct` :

```
//R -> Return type
//Types -> any different type
R operator()(Type name, Type2 name2, ...)
{
    //Do something
    //return something
}

//Use it like this (R is return type, a and b are variables)
R foo = object(a, b, ...);
```

Zum Beispiel:

```
struct Sum
{
    int operator()(int a, int b)
    {
        return a + b;
    }
```

```
};

//Create instance of struct
Sum sum;
int result = sum(1, 1); //result == 2
```

Aufgabenverwalter

Der Zuweisungsoperator ist einer der wichtigsten Operatoren, da Sie den Status einer Variablen ändern können.

Wenn Sie den Assigment-Operator für Ihre `class` / `struct` nicht überladen, wird er automatisch vom Compiler generiert: Der automatisch generierte Zuweisungsoperator führt eine "memberwise-Zuweisung" durch, dh durch Aufrufen von Zuweisungsoperatoren für alle Member, sodass ein Objekt kopiert wird zum anderen ein Mitglied zur Zeit. Der Zuweisungsoperator sollte überlastet sein, wenn die einfache memberweise Zuweisung für Ihre `class` / `struct` nicht geeignet ist, beispielsweise wenn Sie eine **tiefe Kopie** eines Objekts erstellen müssen.

Das Überladen des Zuweisungsoperators `=` ist einfach, aber Sie sollten einige einfache Schritte ausführen.

1. **Testen Sie die Selbstzuweisung.** Diese Überprüfung ist aus zwei Gründen wichtig:
    - Eine Selbstzuweisung ist eine unnötige Kopie, daher macht es keinen Sinn, sie auszuführen.
    - Der nächste Schritt wird bei einer Selbstzuweisung nicht funktionieren.
2. **Bereinigen Sie die alten Daten.** Die alten Daten müssen durch neue ersetzt werden. Nun können Sie den zweiten Grund des vorherigen Schritts verstehen: Wenn der Inhalt des Objekts zerstört wurde, kann die Kopie nicht durch eine Selbstzuweisung ausgeführt werden.
3. **Kopieren Sie alle Mitglieder.** Wenn Sie den Assigment-Operator für Ihre `class` oder Ihre `struct` überladen, wird er nicht automatisch vom Compiler generiert. Sie müssen also alle Member aus dem anderen Objekt kopieren.
4. **Gib** `*this` . Der Operator kehrt per Referenz von selbst zurück, da er die Verkettung ermöglicht (dh `int b = (a = 6) + 4; //b == 10`).

```
//T is some type
T& operator=(const T& other)
{
    //Do something (like copying values)
    return *this;
}
```

Anmerkung: `other` wird von `const&` , da das zugewiesene Objekt nicht geändert werden sollte und die Referenzübergabe schneller ist als durch den Wert. Um sicherzustellen, dass `operator=` es nicht versehentlich ändert, ist es `const` .

Der Zuweisungsoperator kann **nur** in der `class` / `struct` überladen werden, da der linke Wert von `=` **immer** die `class` / `struct` selbst ist. Das Definieren als freie Funktion hat diese Garantie nicht und ist daher nicht zulässig.

Wenn Sie es in der `class` / `struct` deklarieren, ist der linke Wert implizit die `class` / `struct` selbst, sodass dies kein Problem darstellt.

Bitweiser NICHT Operator

Das bitweise NOT (`~`) zu überladen ist ziemlich einfach. *Scrollen Sie zur Erklärung nach unten*

Überladung außerhalb der `class` / `struct` :

```
T operator~(T lhs)
{
    //Do operation
    return lhs;
}
```

Überladung innerhalb von `class` / `struct` :

```
T operator~()
{
    T t(*this);
    //Do operation
    return t;
}
```

Hinweis: Der `operator~` gibt den Wert zurück, da er einen neuen Wert (den geänderten Wert) und nicht einen Verweis auf den Wert zurückgeben muss (dies wäre ein Verweis auf das temporäre Objekt, in dem sich sobald ein Wert für den Wert befindet der Operator ist fertig). Auch nicht `const` da der aufrufende Code ihn nachträglich ändern kann (dh `int a = ~a + 1;` sollte möglich sein).

Innerhalb der `class` / `struct` Sie ein temporäres Objekt erstellen, da Sie dies nicht ändern `this` , da dies das ursprüngliche Objekt ändern würde. `this` sollte jedoch nicht der Fall sein.

Bit-Shift-Operatoren für E / A

Die Operatoren `<<` und `>>` werden üblicherweise als "Write" - und "Read" -Operatoren verwendet:

- `std::ostream` `<<` , um Variablen in den zugrunde liegenden Stream zu schreiben (Beispiel: `std::cout`)
- `std::istream` überladen `>>` , um aus dem zugrunde liegenden Stream eine Variable zu lesen (Beispiel: `std::cin`)

Die Vorgehensweise ist ähnlich, wenn Sie sie "normal" außerhalb der `class` / `struct` überladen `struct` , mit der Ausnahme, dass die Angabe der Argumente nicht vom selben Typ ist:

- Der Rückgabetyp ist der Stream, den Sie überladen möchten (z. B. `std::ostream`), um eine Verkettung zuzulassen (Chaining: `std::cout << a << b;`). Beispiel: `std::ostream&`
- `lhs` wäre das gleiche wie der Rückgabetyp
- `rhs` ist der Typ, für den Sie eine Überladung zulassen möchten (z. B. `T`), der von `const&` anstelle des Werts aus Leistungsgründen übergeben wird (`rhs` sollte sowieso nicht geändert

werden). Beispiel: `const Vector&` .

Beispiel:

```
//Overload std::ostream operator<< to allow output from Vector's
std::ostream& operator<<(std::ostream& lhs, const Vector& rhs)
{
    lhs << "x: " << rhs.x << " y: " << rhs.y << " z: " << rhs.z << '\n';
    return lhs;
}

Vector v = { 1, 2, 3};

//Now you can do
std::cout << v;
```

Komplexe Zahlen werden erneut betrachtet

Mit dem folgenden Code wird ein sehr einfacher komplexer Zahlentyp implementiert, für den das zugrunde liegende Feld gemäß den Regeln für die Typumwandlung der Sprache unter Verwendung der vier Basisoperatoren (+, -, * und /) mit einem Mitglied eines anderen Felds automatisch hochgestuft wird (sei es ein anderer `complex<T>` oder ein anderer Skalartyp).

Dies soll ein ganzheitliches Beispiel sein, das die Überlastung der Bediener und die grundlegende Verwendung von Vorlagen abdeckt.

```
#include <type_traits>

namespace not_std{

using std::decay_t;

//-------------------------------------------------------------------
// complex< value_t >
//-------------------------------------------------------------------

template<typename value_t>
struct complex
{
    value_t x;
    value_t y;

    complex &operator += (const value_t &x)
    {
        this->x += x;
        return *this;
    }
    complex &operator += (const complex &other)
    {
        this->x += other.x;
        this->y += other.y;
        return *this;
    }

    complex &operator -= (const value_t &x)
    {
```

```
        this->x -= x;
        return *this;
    }
    complex &operator -= (const complex &other)
    {
        this->x -= other.x;
        this->y -= other.y;
        return *this;
    }

    complex &operator *= (const value_t &s)
    {
        this->x *= s;
        this->y *= s;
        return *this;
    }
    complex &operator *= (const complex &other)
    {
        (*this) = (*this) * other;
        return *this;
    }

    complex &operator /= (const value_t &s)
    {
        this->x /= s;
        this->y /= s;
        return *this;
    }
    complex &operator /= (const complex &other)
    {
        (*this) = (*this) / other;
        return *this;
    }

    complex(const value_t &x, const value_t &y)
    : x{x}
    , y{y}
    {}

    template<typename other_value_t>
    explicit complex(const complex<other_value_t> &other)
    : x{static_cast<const value_t &>(other.x)}
    , y{static_cast<const value_t &>(other.y)}
    {}

    complex &operator = (const complex &) = default;
    complex &operator = (complex &&) = default;
    complex(const complex &) = default;
    complex(complex &&) = default;
    complex() = default;
};

// Absolute value squared
template<typename value_t>
value_t absqr(const complex<value_t> &z)
{ return z.x*z.x + z.y*z.y; }

//---------------------------------------------------------------
// operator - (negation)
//---------------------------------------------------------------
```

```
template<typename value_t>
complex<value_t> operator - (const complex<value_t> &z)
{ return {-z.x, -z.y}; }

//-------------------------------------------------------------------
// operator +
//-------------------------------------------------------------------

template<typename left_t,typename right_t>
auto operator + (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x + b.x)>>
{ return{a.x + b.x, a.y + b.y}; }

template<typename left_t,typename right_t>
auto operator + (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a + b.x)>>
{ return{a + b.x, b.y}; }

template<typename left_t,typename right_t>
auto operator + (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x + b)>>
{ return{a.x + b, a.y}; }

//-------------------------------------------------------------------
// operator -
//-------------------------------------------------------------------

template<typename left_t,typename right_t>
auto operator - (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x - b.x)>>
{ return{a.x - b.x, a.y - b.y}; }

template<typename left_t,typename right_t>
auto operator - (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a - b.x)>>
{ return{a - b.x, - b.y}; }

template<typename left_t,typename right_t>
auto operator - (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x - b)>>
{ return{a.x - b, a.y}; }

//-------------------------------------------------------------------
// operator *
//-------------------------------------------------------------------

template<typename left_t, typename right_t>
auto operator * (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x * b.x)>>
{
    return {
        a.x*b.x - a.y*b.y,
        a.x*b.y + a.y*b.x
        };
}

template<typename left_t, typename right_t>
auto operator * (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a * b.x)>>
{ return {a * b.x, a * b.y}; }
```

```
template<typename left_t, typename right_t>
auto operator * (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x * b)>>
{ return {a.x * b, a.y * b}; }

//-----------------------------------------------------------------
// operator /
//-----------------------------------------------------------------

template<typename left_t, typename right_t>
auto operator / (const complex<left_t> &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a.x / b.x)>>
{
    const auto r = absqr(b);
    return {
        ( a.x*b.x + a.y*b.y) / r,
        (-a.x*b.y + a.y*b.x) / r
        };
}

template<typename left_t, typename right_t>
auto operator / (const left_t &a, const complex<right_t> &b)
-> complex<decay_t<decltype(a / b.x)>>
{
    const auto s = a/absqr(b);
    return {
         b.x * s,
        -b.y * s
        };
}

template<typename left_t, typename right_t>
auto operator / (const complex<left_t> &a, const right_t &b)
-> complex<decay_t<decltype(a.x / b)>>
{ return {a.x / b, a.y / b}; }

}// namespace not_std


int main(int argc, char **argv)
{
    using namespace not_std;

    complex<float> fz{4.0f, 1.0f};

    // makes a complex<double>
    auto dz = fz * 1.0;

    // still a complex<double>
    auto idz = 1.0f/dz;

    // also a complex<double>
    auto one = dz * idz;

    // a complex<double> again
    auto one_again = fz * idz;

    // Operator tests, just to make sure everything compiles.

    complex<float> a{1.0f, -2.0f};
    complex<double> b{3.0, -4.0};
```

```
    // All of these are complex<double>
    auto c0 = a + b;
    auto c1 = a - b;
    auto c2 = a * b;
    auto c3 = a / b;

    // All of these are complex<float>
    auto d0 = a + 1;
    auto d1 = 1 + a;
    auto d2 = a - 1;
    auto d3 = 1 - a;
    auto d4 = a * 1;
    auto d5 = 1 * a;
    auto d6 = a / 1;
    auto d7 = 1 / a;

    // All of these are complex<double>
    auto e0 = b + 1;
    auto e1 = 1 + b;
    auto e2 = b - 1;
    auto e3 = 1 - b;
    auto e4 = b * 1;
    auto e5 = 1 * b;
    auto e6 = b / 1;
    auto e7 = 1 / b;

    return 0;
}
```

Benannte Betreiber

Sie können C ++ mit benannten Operatoren erweitern, die von Standard-C ++ - Operatoren "in Anführungszeichen" gesetzt werden.

Zuerst beginnen wir mit einer Dutzend-Zeilen-Bibliothek:

```
namespace named_operator {
  template<class D>struct make_operator{constexpr make_operator(){}};

  template<class T, char, class O> struct half_apply { T&& lhs; };

  template<class Lhs, class Op>
  half_apply<Lhs, '*', Op> operator*( Lhs&& lhs, make_operator<Op> ) {
    return {std::forward<Lhs>(lhs)};
  }

  template<class Lhs, class Op, class Rhs>
  auto operator*( half_apply<Lhs, '*', Op>&& lhs, Rhs&& rhs )
  -> decltype( named_invoke( std::forward<Lhs>(lhs.lhs), Op{}, std::forward<Rhs>(rhs) ) )
  {
    return named_invoke( std::forward<Lhs>(lhs.lhs), Op{}, std::forward<Rhs>(rhs) );
  }
}
```

das tut noch nichts.

Zuerst werden Vektoren angehängt

```
namespace my_ns {
  struct append_t : named_operator::make_operator<append_t> {};
  constexpr append_t append{};

  template<class T, class A0, class A1>
  std::vector<T, A0> named_invoke( std::vector<T, A0> lhs, append_t, std::vector<T, A1> const&
rhs ) {
      lhs.insert( lhs.end(), rhs.begin(), rhs.end() );
      return std::move(lhs);
  }
}
using my_ns::append;

std::vector<int> a {1,2,3};
std::vector<int> b {4,5,6};

auto c = a *append* b;
```

Der Kern hier ist, dass wir ein `append` Objekt vom Typ `append_t:named_operator::make_operator<append_t>` .

Dann überladen wir named_invoke (lhs, append_t, rhs) für die Typen, die wir rechts und links möchten.

Die Bibliothek überlastet `lhs*append_t` und gibt ein temporäres `half_apply` Objekt zurück. Außerdem wird `half_apply*rhs` überladen, um `named_invoke( lhs, append_t, rhs )` .

Wir müssen einfach das richtige `append_t` Token erstellen und eine ADL-freundliche `named_invoke` mit der richtigen Signatur `named_invoke` , und alles hängt zusammen und funktioniert.

Angenommen, Sie möchten eine elementweise Multiplikation der Elemente eines std :: -Arrays haben:

```
template<class=void, std::size_t...Is>
auto indexer( std::index_sequence<Is...> ) {
  return [](auto&& f) {
    return f( std::integral_constant<std::size_t, Is>{}... );
  };
}
template<std::size_t N>
auto indexer() { return indexer( std::make_index_sequence<N>{} ); }

namespace my_ns {
  struct e_times_t : named_operator::make_operator<e_times_t> {};
  constexpr e_times_t e_times{};

  template<class L, class R, std::size_t N,
    class Out=std::decay_t<decltype( std::declval<L const&>()*std::declval<R const&>() )>
  >
  std::array<Out, N> named_invoke( std::array<L, N> const& lhs, e_times_t, std::array<R, N>
const& rhs ) {
    using result_type = std::array<Out, N>;
    auto index_over_N = indexer<N>();
    return index_over_N([&](auto...is)->result_type {
```

```
      return {{
        (lhs[is] * rhs[is])...
      }};
    });
  }
}
```

Live-Beispiel .

Dieser elementweise Arraycode kann für Tupel oder Paare oder Arrays im C-Stil oder sogar für Container mit variabler Länge verwendet werden, wenn Sie entscheiden, was zu tun ist, wenn die Längen nicht übereinstimmen.

Sie könnten auch einen elementweisen Operator-Typ `lhs *element_wise<'+'>* rhs` und `lhs *element_wise<'+'>* rhs` .

Das Schreiben von `*dot*` und `*cross*` -Produktoperatoren ist ebenfalls eine offensichtliche Verwendung.

Die Verwendung von `*` kann erweitert werden, um andere Trennzeichen wie `+` . Die Delimetermeßgenauigkeit bestimmt die Genauigkeit des genannten Operators. Dies kann wichtig sein, wenn Physikgleichungen mit minimalem Einsatz von extra `()` s nach C ++ übersetzt werden.

Mit einer geringfügigen Änderung in der Bibliothek können wir `->*then*` -Operatoren unterstützen und die `std::function` vor der Aktualisierung des Standards erweitern oder monadisch schreiben `->*bind*` . Es könnte auch einen stateful-benannten Operator haben, bei dem wir das `Op` vorsichtig an die endgültige Aufruffunktion übergeben, was Folgendes ermöglicht:

```
named_operator<'*'> append = [](auto lhs, auto&& rhs) {
  using std::begin; using std::end;
  lhs.insert( end(lhs), begin(rhs), end(rhs) );
  return std::move(lhs);
};
```

einen benannten Container anhängenden Operator in C ++ 17 erzeugen.

Überladung des Bedieners online lesen: https://riptutorial.com/de/cplusplus/topic/562/uberladung-des-bedieners

Kapitel 132: Überlastauflösung

Bemerkungen

Die Überlastauflösung tritt in verschiedenen Situationen auf

- Ruft benannte überladene Funktionen auf. Die Kandidaten sind alle Funktionen, die durch Namenssuche gefunden werden.
- Aufrufe des Klassenobjekts Die Kandidaten sind normalerweise alle überladenen Funktionsaufrufoperatoren der Klasse.
- Verwendung eines Operators Die Kandidaten sind die überladenen Operatorfunktionen im Namespace-Bereich, die überladenen Operatorfunktionen im linken Klassenobjekt (falls vorhanden) und die integrierten Operatoren.
- Überladungsauflösung, um die korrekte Konvertierungsoperatorfunktion oder den Konstruktor für eine Initialisierung aufzurufen
    - Bei der Direktinitialisierung außerhalb der Liste (`Class c(value)`) handelt es sich bei den Kandidaten um Konstruktoren von `Class` .
    - Für die Nicht-Listenkopie-Initialisierung (`Class c = value`) und für das Auffinden der benutzerdefinierten Konvertierungsfunktion, die in einer benutzerdefinierten Konvertierungssequenz aufgerufen werden soll. Die Kandidaten sind die Konstruktoren von `Class` Wenn die Quelle ein Klassenobjekt ist, fungiert der Konvertierungsoperator.
    - Zur Initialisierung einer Nicht-Klasse aus einem Klassenobjekt (Nicht-Klasse `Nonclass c = classObject`). Die Kandidaten sind die Konvertierungsoperatorfunktionen des Initialisierungsobjekts.
    - Zum Initialisieren einer Referenz mit einem Klassenobjekt (`R &r = classObject`), wenn die Klasse über Konvertierungsoperatorfunktionen verfügt, die Werte liefern, die direkt an `r` gebunden werden können. Die Kandidaten sind solche Konvertierungsoperatorfunktionen.
    - Bei der Listeninitialisierung eines nicht aggregierten Klassenobjekts (`Class c{1, 2, 3}`) sind die Kandidaten die Initialisierungslistenkonstruktoren für eine Auflösung des ersten Durchlaufs durch Überladung. Wenn dies keinen geeigneten Kandidaten findet, wird eine zweite Überlastungslösung mit den Konstruktoren von `Class` als Kandidaten durchgeführt.

Examples

Genaue Übereinstimmung

Eine Überladung ohne Konvertierungen, die für Parametertypen erforderlich sind, oder nur Konvertierungen, die zwischen Typen bestehen, die noch als exakte Übereinstimmungen betrachtet werden, wird gegenüber einer Überladung bevorzugt, für deren Aufruf andere Konvertierungen erforderlich sind.

```
void f(int x);
void f(double x);
```

```
f(42); // calls f(int)
```

Wenn ein Argument an einen Verweis auf den gleichen Typ gebunden wird, wird davon ausgegangen, dass die Übereinstimmung keine Konvertierung erfordert, selbst wenn der Verweis mehr Cv-qualifiziert ist.

```
void f(int& x);
void f(double x);
int x = 42;
f(x); // argument type is int; exact match with int&

void g(const int& x);
void g(int x);
g(x); // ambiguous; both overloads give exact match
```

Zum Zwecke der Überladungsauflösung wird davon ausgegangen, dass der Typ "Array von `T` " genau mit dem Typ "Zeiger auf `T` " übereinstimmt, und der Funktionstyp `T` stimmt mit dem Funktionszeigertyp `T*` genau überein, obwohl beide dies erfordern Konvertierungen.

```
void f(int* p);
void f(void* p);

void g(int* p);
void g(int (&p)[100]);

int a[100];
f(a); // calls f(int*); exact match with array-to-pointer conversion
g(a); // ambiguous; both overloads give exact match
```

Kategorisierung von Argumenten zu Parameterkosten

Die Überladungsauflösung partitioniert die Kosten für die Übergabe eines Arguments an einen Parameter in eine von vier verschiedenen Kategorien, die als "Sequenzen" bezeichnet werden. Jede Sequenz kann null, eine oder mehrere Konvertierungen enthalten

- Standard-Konvertierungssequenz

  ```
  void f(int a); f(42);
  ```

- Benutzerdefinierte Konvertierungssequenz

  ```
  void f(std::string s); f("hello");
  ```

- Ellipsis-Konvertierungssequenz

  ```
  void f(...); f(42);
  ```

- Initialisierungssequenz auflisten

  ```
  void f(std::vector<int> v); f({1, 2, 3});
  ```

Das allgemeine Prinzip ist, dass Standard-Konvertierungssequenzen am billigsten sind, gefolgt von benutzerdefinierten Konvertierungssequenzen, gefolgt von Ellipsis-Konvertierungssequenzen.

Ein Sonderfall ist die Listeninitialisierungssequenz, die keine Konvertierung darstellt (eine Initialisierungsliste ist kein Ausdruck mit einem Typ). Ihre Kosten werden bestimmt, indem definiert wird, dass sie einer der anderen drei Konvertierungssequenzen entspricht, abhängig vom Parametertyp und der Form der Initialisierungsliste.

Namenssuche und Zugriffsprüfung

Die Überlastauflösung erfolgt *nach der* Namenssuche. Das bedeutet, dass eine besser passende Funktion nicht durch Überladungsauflösung ausgewählt wird, wenn die Namenssuche verloren geht:

```
void f(int x);
struct S {
    void f(double x);
    void g() { f(42); } // calls S::f because global f is not visible here,
                        // even though it would be a better match
};
```

Die Überlastauflösung erfolgt *vor der* Zugriffsprüfung. Eine nicht zugreifbare Funktion kann durch Überlastauflösung ausgewählt werden, wenn sie besser als eine zugreifbare Funktion ist.

```
class C {
  public:
    static void f(double x);
  private:
    static void f(int x);
};
C::f(42); // Error! Calls private C::f(int) even though public C::f(double) is viable.
```

In ähnlicher Weise erfolgt die Überlastlösung, ohne zu prüfen, ob der resultierende Aufruf in Bezug auf `explicit` :

```
struct X {
    explicit X(int );
    X(char );
};

void foo(X );
foo({4}); // X(int) is better much, but expression is
          // ill-formed because selected constructor is explicit
```

Überladen der Weiterleitungsreferenz

Sie müssen beim Übermitteln einer Weiterleitungsreferenz sehr vorsichtig sein, da dies möglicherweise zu gut passt:

```
struct A {
    A() = default;           // #1
```

```
    A(A const& ) = default;  // #2

    template <class T>
    A(T&& );                 // #3
};
```

Die Absicht hier war, dass `A` kopierbar ist und dass wir diesen anderen Konstruktor haben, der ein anderes Member initialisieren könnte. Jedoch:

```
A a;     // calls #1
A b(a);  // calls #3!
```

Es gibt zwei mögliche Übereinstimmungen für den Bauaufruf:

```
A(A const& ); // #2
A(A& );       // #3, with T = A&
```

Beide sind genaue Übereinstimmung, aber `#3` nimmt einen Verweis auf eine weniger *cv* , Qualifizierte Objekt als `#2` der Fall ist, so hat es die bessere Standardumwandlungsfolge und ist die beste tragfähige Funktion.

Die Lösung hier ist, diese Konstruktoren immer einzuschränken (z. B. mit SFINAE):

```
template <class T,
    class = std::enable_if_t<!std::is_convertible<std::decay_t<T>*, A*>::value>
    >
A(T&& );
```

Das Typmerkmal besteht darin, ein beliebiges `A` oder eine Klasse, die öffentlich und eindeutig von `A` abgeleitet ist, von der Betrachtung auszuschließen, was dazu führen würde, dass dieser Konstruktor in dem zuvor beschriebenen Beispiel falsch geformt wurde (und daher aus dem Überlastungssatz entfernt wurde). Als Ergebnis wird der Kopierkonstruktor aufgerufen - was wir wollten.

Schritte zur Überlastauflösung

Die Schritte der Überlastlösung sind:

1. Finden Sie Kandidatenfunktionen über die Namenssuche. Unqualifizierte Aufrufe führen sowohl eine regelmäßige unqualifizierte Suche als auch eine argumentabhängige Suche (falls zutreffend) durch.

2. Filtern Sie die Menge der möglichen Funktionen nach einer Reihe *praktikabler* Funktionen. Eine realisierbare Funktion, für die eine implizite Konvertierungssequenz zwischen den Argumenten, mit denen die Funktion aufgerufen wird, und den von der Funktion verwendeten Parametern vorhanden ist.

   ```
   void f(char);          // (1)
   void f(int ) = delete; // (2)
   ```

```
void f();              // (3)
void f(int& );         // (4)

f(4); // 1,2 are viable (even though 2 is deleted!)
      // 3 is not viable because the argument lists don't match
      // 4 is not viable because we cannot bind a temporary to
      //     a non-const lvalue reference
```

3. Wählen Sie den besten Kandidaten aus. Eine realisierbare Funktion `F1` ist eine bessere Funktion als eine andere realisierbare Funktion `F2` wenn die implizite Konvertierungssequenz für jedes Argument in `F1` nicht schlechter als die entsprechende implizite Konvertierungssequenz in `F2` ist und ...:

   3.1. Für einige Argumente ist die implizite Konvertierungssequenz für dieses Argument in `F1` eine bessere Konvertierungssequenz als für dieses Argument in `F2` oder

```
void f(int );  // (1)
void f(char ); // (2)

f(4);  // call (1), better conversion sequence
```

   3.2. Bei einer benutzerdefinierten Konvertierung ist die Standardkonvertierungssequenz von der Rückkehr von `F1` zum Zieltyp eine bessere Konvertierungssequenz als die des Rückgabetyps von `F2` oder

```
struct A
{
    operator int();
    operator double();
} a;

int i = a; // a.operator int() is better than a.operator double() and a conversion
float f = a; // ambiguous
```

   3.3. In einer direkten Referenzbindung hat `F1` die gleiche Art von Referenz von `F2` nicht oder nicht

```
struct A
{
    operator X&();  // #1
    operator X&&(); // #2
};
A a;
X& lx = a;  // calls #1
X&& rx = a; // calls #2
```

   3.4. `F1` ist keine Funktionsvorlagenspezialisierung, aber `F2` ist oder

```
template <class T> void f(T ); // #1
void f(int );                  // #2

f(42); // calls #2, the non-template
```

3,5. `F1` und `F2` sind beide Funktionsschablonenspezialisierungen, aber `F1` ist spezialisierter als `F2` .

```
template <class T> void f(T );  // #1
template <class T> void f(T* ); // #2

int* p;
f(p); // calls #2, more specialized
```

Die Reihenfolge hier ist signifikant. Die bessere Prüfung der Konvertierungssequenz findet vor der Vorlage statt der Prüfung ohne Vorlage statt. Dies führt zu einem allgemeinen Fehler bei der Überladung der Weiterleitungsreferenz:

```
struct A {
    A(A const& ); // #1

    template <class T>
    A(T&& );      // #2, not constrained
};

A a;
A b(a); // calls #2!
        // #1 is not a template but #2 resolves to
        // A(A& ), which is a less cv-qualified reference than #1
        // which makes it a better implicit conversion sequence
```

Wenn am Ende kein einziger bester Kandidat vorhanden ist, ist der Anruf mehrdeutig:

```
void f(double ) { }
void f(float ) { }

f(42); // error: ambiguous
```

Arithmetische Promotionen und Konvertierungen

Das Konvertieren eines Integer-Typs in den entsprechenden heraufgestuften Typ ist besser als das Konvertieren in einen anderen Integer-Typ.

```
void f(int x);
void f(short x);
signed char c = 42;
f(c); // calls f(int); promotion to int is better than conversion to short
short s = 42;
f(s); // calls f(short); exact match is better than promotion to int
```

Es ist besser, einen `float` in das `double` , als ihn in einen anderen Gleitkommatyp zu konvertieren.

```
void f(double x);
void f(long double x);
f(3.14f); // calls f(double); promotion to double is better than conversion to long double
```

Andere arithmetische Konvertierungen als Beförderungen sind weder besser noch schlechter als einander.

```
void f(float x);
void f(long double x);
f(3.14); // ambiguous

void g(long x);
void g(long double x);
g(42); // ambiguous
g(3.14); // ambiguous
```

Um sicherzustellen, dass beim Aufruf einer Funktion `f` mit Integral- oder Gleitkomma-Argumenten eines beliebigen Standardtyps keine Mehrdeutigkeit auftritt, sind insgesamt acht Überladungen erforderlich, sodass für jeden möglichen Argumenttyp entweder eine Überladung passt genau oder die eindeutige Überladung mit dem beförderten Argumenttyp wird ausgewählt.

```
void f(int x);
void f(unsigned int x);
void f(long x);
void f(unsigned long x);
void f(long long x);
void f(unsigned long long x);
void f(double x);
void f(long double x);
```

Überladen innerhalb einer Klassenhierarchie

In den folgenden Beispielen wird diese Klassenhierarchie verwendet:

```
struct A { int m; };
struct B : A {};
struct C : B {};
```

Die Konvertierung vom abgeleiteten Klassentyp in den Basisklassentyp wird gegenüber benutzerdefinierten Konvertierungen bevorzugt. Dies gilt bei der Übergabe als Wert oder Referenz, sowie bei der Konvertierung von Zeiger in abgeleitete in Zeiger in Basis.

```
struct Unrelated {
    Unrelated(B b);
};
void f(A a);
void f(Unrelated u);
B b;
f(b); // calls f(A)
```

Eine Zeigerkonvertierung von einer abgeleiteten Klasse in eine Basisklasse ist auch besser als eine Konvertierung in `void*` .

```
void f(A* p);
void f(void* p);
B b;
```

```
f(&b); // calls f(A*)
```

Wenn mehrere Überladungen in derselben Vererbungskette vorhanden sind, wird die am stärksten abgeleitete Basisklassenüberladung bevorzugt. Dies basiert auf einem ähnlichen Prinzip wie der virtuelle Versand: Die "spezialisierteste" Implementierung wird ausgewählt. Die Überladungsauflösung tritt jedoch immer zur Kompilierzeit auf und wird niemals implizit herabgesetzt.

```
void f(const A& a);
void f(const B& b);
C c;
f(c); // calls f(const B&)
B b;
A& r = b;
f(r); // calls f(const A&); the f(const B&) overload is not viable
```

Für Zeiger auf Member, die der Klasse zuwiderlaufen, gilt eine ähnliche Regel in umgekehrter Richtung: Die am wenigsten abgeleitete abgeleitete Klasse wird bevorzugt.

```
void f(int B::*p);
void f(int C::*p);
int A::*p = &A::m;
f(p); // calls f(int B::*)
```

Überlastung von Konstanz und Volatilität

Die Übergabe eines Zeigerarguments an einen `T*` -Parameter ist, wenn möglich, besser als die Übergabe an einen `const T*` -Parameter.

```
struct Base {};
struct Derived : Base {};
void f(Base* pb);
void f(const Base* pb);
void f(const Derived* pd);
void f(bool b);

Base b;
f(&b); // f(Base*) is better than f(const Base*)
Derived d;
f(&d); // f(const Derived*) is better than f(Base*) though;
       // constness is only a "tie-breaker" rule
```

Ebenso ist das Übergeben eines Arguments an einen `T&` Parameter, wenn möglich, besser als ein Übergeben an einen `const T&` Parameter, auch wenn beide einen genauen Übereinstimmungsrang haben.

```
void f(int& r);
void f(const int& r);
int x;
f(x); // both overloads match exactly, but f(int&) is still better
const int y = 42;
f(y); // only f(const int&) is viable
```

Diese Regel gilt auch für const-qualifizierte Memberfunktionen, bei denen es wichtig ist, den veränderbaren Zugriff auf Nicht-const-Objekte und den unveränderlichen Zugriff auf const-Objekte zuzulassen.

```
class IntVector {
  public:
    // ...
    int* data() { return m_data; }
    const int* data() const { return m_data; }
  private:
    // ...
    int* m_data;
};
IntVector v1;
int* data1 = v1.data();       // Vector::data() is better than Vector::data() const;
                              // data1 can be used to modify the vector's data
const IntVector v2;
const int* data2 = v2.data(); // only Vector::data() const is viable;
                              // data2 can't be used to modify the vector's data
```

In gleicher Weise wird eine flüchtige Überlastung weniger bevorzugt als eine nichtflüchtige Überlastung.

```
class AtomicInt {
  public:
    // ...
    int load();
    int load() volatile;
  private:
    // ...
};
AtomicInt a1;
a1.load(); // non-volatile overload preferred; no side effect
volatile AtomicInt a2;
a2.load(); // only volatile overload is viable; side effect
static_cast<volatile AtomicInt&>(a1).load(); // force volatile semantics for a1
```

Überlastauflösung online lesen: https://riptutorial.com/de/cplusplus/topic/2021/uberlastauflosung

Kapitel 133: Unbekanntes Verhalten

Bemerkungen

Wenn das Verhalten eines Konstrukts nicht spezifiziert ist, gibt der Standard einige Einschränkungen für das Verhalten vor, lässt jedoch einige Freiräume für die Implementierung, was *nicht* erforderlich ist, um zu dokumentieren, was in einer bestimmten Situation geschieht. Sie steht im Gegensatz zum implementierungsdefinierten Verhalten , bei dem die Implementierung erforderlich *ist* , um zu dokumentieren, was passiert, und undefiniertes Verhalten, bei dem alles passieren kann.

Examples

Reihenfolge der Initialisierung von Globals in der gesamten TU

Während in einer Übersetzungseinheit die Reihenfolge der Initialisierung von globalen Variablen festgelegt wird, ist die Reihenfolge der Initialisierung über die Übersetzungseinheiten nicht spezifiziert.

Also mit folgenden Dateien programmieren

- foo.cpp

```
#include <iostream>

int dummyFoo = ((std::cout << "foo"), 0);
```

- bar.cpp

```
#include <iostream>

int dummyBar = ((std::cout << "bar"), 0);
```

- main.cpp

```
int main() {}
```

kann als Ausgabe produzieren:

```
foobar
```

oder

```
barfoo
```

Dies kann zu einem *Fiasko* für *statische Initialisierung* führen.

Wert einer Aufzählung außerhalb des Bereichs

Wenn eine bereichsabhängige Aufzählung in einen ganzzahligen Typ umgewandelt wird, der zu klein ist, um den Wert zu halten, ist der resultierende Wert nicht angegeben. Beispiel:

```
enum class E {
    X = 1,
    Y = 1000,
};
// assume 1000 does not fit into a char
char c1 = static_cast<char>(E::X); // c1 is 1
char c2 = static_cast<char>(E::Y); // c2 has an unspecified value
```

Wenn eine Ganzzahl in eine Aufzählung umgewandelt wird und der Wert der Ganzzahl außerhalb des Wertebereichs der Aufzählung liegt, ist der resultierende Wert nicht angegeben. Beispiel:

```
enum Color {
    RED = 1,
    GREEN = 2,
    BLUE = 3,
};
Color c = static_cast<Color>(4);
```

Im nächsten Beispiel ist das Verhalten jedoch *nicht* unbestimmt, da der Quellwert innerhalb des *Bereichs* der Enumeration liegt, obwohl er nicht allen Enumeratoren entspricht:

```
enum Scale {
    ONE = 1,
    TWO = 2,
    FOUR = 4,
};
Scale s = static_cast<Scale>(3);
```

Hier wird `s` den Wert 3 haben und ungleich `ONE` , `TWO` und `FOUR` .

Statischer Wurf aus falschem void * -Wert

Wenn ein `void*` -Wert in einen Zeiger auf den Objekttyp `T*` konvertiert wird, für `T` jedoch nicht ordnungsgemäß ausgerichtet ist, ist der resultierende Zeigerwert nicht angegeben. Beispiel:

```
// Suppose that alignof(int) is 4
int x = 42;
void* p1 = &x;
// Do some pointer arithmetic...
void* p2 = static_cast<char*>(p1) + 2;
int* p3 = static_cast<int*>(p2);
```

Der Wert von `p3` ist nicht angegeben, da `p2` nicht auf ein Objekt vom Typ `int` . Ihr Wert ist keine richtig ausgerichtete Adresse.

Ergebnis einiger reinterpret_cast-Konvertierungen

Das Ergebnis eines `reinterpret_cast` von einem Funktionszeigertyp zu einem anderen oder eines Funktionsreferenztyps zu einem anderen ist nicht angegeben. Beispiel:

```
int f();
auto fp = reinterpret_cast<int(*)(int)>(&f); // fp has unspecified value
```

C ++ 03

Das Ergebnis eines `reinterpret_cast` von einem Objektzeigertyp zu einem anderen oder von einem Objektreferenztyp zu einem anderen ist nicht angegeben. Beispiel:

```
int x = 42;
char* p = reinterpret_cast<char*>(&x); // p has unspecified value
```

Bei den meisten Compilern entsprach dies jedoch `static_cast<char*>(static_cast<void*>(&x))` sodass der resultierende Zeiger `p` auf das erste Byte von `x` . Dies wurde zum Standardverhalten in C ++ 11. Weitere Informationen finden Sie unter Typ-Punning-Konvertierung .

Ergebnis einiger Zeigervergleiche

Wenn zwei Zeiger mit `<` , `>` , `<=` oder `>=` verglichen werden, ist das Ergebnis in den folgenden Fällen nicht angegeben:

- Die Zeiger verweisen auf verschiedene Arrays. (Ein Nicht-Array-Objekt wird als Array der Größe 1 betrachtet.)

```
int x;
int y;
const bool b1 = &x < &y;            // unspecified
int a[10];
const bool b2 = &a[0] < &a[1];      // true
const bool b3 = &a[0] < &x;         // unspecified
const bool b4 = (a + 9) < (a + 10); // true
                                    // note: a+10 points past the end of the array
```

- Die Zeiger verweisen auf dasselbe Objekt, jedoch auf Mitglieder mit unterschiedlicher Zugriffskontrolle.

```
class A {
  public:
    int x;
    int y;
    bool f1() { return &x < &y; } // true; x comes before y
    bool f2() { return &x < &z; } // unspecified
  private:
    int z;
};
```

Platz, der von einer Referenz belegt wird

Eine Referenz ist kein Objekt und im Gegensatz zu einem Objekt kann nicht garantiert werden,

dass einige zusammenhängende Speicherbytes belegt werden. Der Standard lässt keine Angabe, ob eine Referenz überhaupt Speicherplatz benötigt. Eine Reihe von Merkmalen der Sprache verschwören, um die portierbare Überprüfung des von der Referenz belegten Speichers unmöglich zu machen:

- Wenn `sizeof` auf eine Referenz angewendet wird, wird die Größe des referenzierten Typs zurückgegeben, sodass keine Informationen darüber erhalten werden, ob die Referenz Speicherplatz belegt.
- Arrays von Referenzen sind unzulässig, daher ist es nicht möglich, die Adressen zweier aufeinanderfolgender Elemente einer hypothetischen Referenz von Arrays zu untersuchen, um die Größe einer Referenz zu bestimmen.
- Wenn die Adresse eines Verweises verwendet wird, ist das Ergebnis die Adresse des Verweises, sodass wir keinen Verweis auf den Verweis selbst erhalten können.
- Wenn eine Klasse über ein Referenzelement verfügt, wird beim Versuch, die Adresse dieses `offsetof` mithilfe von `offsetof` zu extrahieren, undefiniertes Verhalten `offsetof` da eine solche Klasse keine Standardlayoutklasse ist.
- Wenn eine Klasse über ein Referenzelement verfügt, ist die Klasse nicht mehr das Standardlayout. Daher wird versucht, auf alle Daten zuzugreifen, die zum Speichern der Referenzergebnisse verwendet werden, undefiniertes oder nicht angegebenes Verhalten.

In der Praxis kann in manchen Fällen eine Referenzvariable ähnlich wie eine Zeigervariable implementiert werden und somit dieselbe Menge an Speicher wie ein Zeiger einnehmen, während in anderen Fällen eine Referenz überhaupt keinen Platz beansprucht, da sie optimiert werden kann. Zum Beispiel in:

```
void f() {
    int x;
    int& r = x;
    // do something with r
}
```

Der Compiler kann `r` einfach als Alias für `x` und alle Vorkommen von `r` im Rest der Funktion `f` durch `x` ersetzen und keinen Speicherplatz für `r` zuweisen.

Bewertungsreihenfolge von Funktionsargumenten

Wenn eine Funktion mehrere Argumente hat, ist nicht angegeben, in welcher Reihenfolge sie ausgewertet werden. Der folgende Code könnte `x = 1, y = 2` oder `x = 2, y = 1` drucken `x = 2, y = 1` aber es ist nicht angegeben, welche.

```
int f(int x, int y) {
    printf("x = %d, y = %d\n", x, y);
}
int get_val() {
    static int x = 0;
    return ++x;
}
int main() {
    f(get_val(), get_val());
}
```

C ++ 17

In C ++ 17 ist die Reihenfolge der Auswertung von Funktionsargumenten nicht spezifiziert.

Jedes Funktionsargument wird jedoch vollständig ausgewertet, und das aufrufende Objekt wird garantiert vor allen Funktionsargumenten ausgewertet.

```
struct from_int {
  from_int(int x) { std::cout << "from_int (" << x << ")\n"; }
};
int make_int(int x){ std::cout << "make_int (" << x << ")\n"; return x; }


void foo(from_int a, from_int b) {
}
void bar(from_int a, from_int b) {
}

auto which_func(bool b){
  std::cout << b?"foo":"bar" << "\n";
  return b?foo:bar;
}

int main(int argc, char const*const* argv) {
  which_func( true )( make_int(1), make_int(2) );
}
```

das muss drucken:

```
bar
make_int(1)
from_int(1)
make_int(2)
from_int(2)
```

oder

```
bar
make_int(2)
from_int(2)
make_int(1)
from_int(1)
```

Es kann *keine* `bar` nach einer der `make` oder `from` 's gedruckt werden und es wird möglicherweise nicht gedruckt:

```
bar
make_int(2)
make_int(1)
from_int(2)
from_int(1)
```

oder ähnliches. Vor der C ++ 17 Druck `bar` nach `make_int` s legal war, wie es beide tun `make_int` s zu tun , alle früheren `from_int` s.

Status der meisten Standard-Bibliotheksklassen verschoben

C ++ 11

Alle Standard-Bibliothekscontainer verbleiben nach dem Verschieben in einem *gültigen, aber nicht angegebenen* Zustand. Im folgenden Code enthält `v2` nach der Verschiebung beispielsweise `{1, 2, 3, 4}` ist jedoch nicht garantiert, dass `v1` leer ist.

```
int main() {
    std::vector<int> v1{1, 2, 3, 4};
    std::vector<int> v2 = std::move(v1);
}
```

Für einige Klassen gibt es einen genau definierten Abzugsstatus. Der wichtigste Fall ist der von `std::unique_ptr<T>` , der nach dem `std::unique_ptr<T>` garantiert null ist.

Unbekanntes Verhalten online lesen: https://riptutorial.com/de/cplusplus/topic/4939/unbekanntes-verhalten

Kapitel 134: Unbenannte Typen

Examples

Unbenannte Klassen

Im Gegensatz zu einer benannten Klasse oder Struktur müssen unbenannte Klassen und Strukturen dort definiert werden, wo sie definiert sind, und können keine Konstruktoren oder Destruktoren enthalten.

```
struct {
    int foo;
    double bar;
} foobar;

foobar.foo = 5;
foobar.bar = 4.0;

class {
    int baz;
public:
    int buzz;

    void setBaz(int v) {
        baz = v;
    }
} barbar;

barbar.setBaz(15);
barbar.buzz = 2;
```

Anonyme Mitglieder

Als nicht standardmäßige Erweiterung für C ++ ermöglichen gängige Compiler die Verwendung von Klassen als anonyme Member.

```
struct Example {
    struct {
        int inner_b;
    };

    int outer_b;

    //The anonymous struct's members are accessed as if members of the parent struct
    Example() : inner_b(2), outer_b(4) {
        inner_b = outer_b + 2;
    }
};

Example ex;

//The same holds true for external code referencing the struct
ex.inner_b -= ex.outer_b;
```

Als Typ-Alias

Unbenannte Klassentypen können auch verwendet werden , wenn Typ - Aliasnamen erstellen, dh über `typedef` und `using` :

C ++ 11

```
using vec2d = struct {
    float x;
    float y;
};
```

```
typedef struct {
    float x;
    float y;
} vec2d;
```

```
vec2d pt;
pt.x = 4.f;
pt.y = 3.f;
```

Anonyme Union

Mitgliedsnamen einer anonymen Gewerkschaft gehören zum Geltungsbereich der Gewerkschaftserklärung und müssen sich von allen anderen Namen dieses Geltungsbereichs unterscheiden. Das Beispiel hier hat den gleichen Aufbau wie das Beispiel Anonyme Mitglieder , die "struct" verwenden, ist jedoch standardkonform.

```
struct Sample {
    union {
        int a;
        int b;
    };
    int c;
};
int main()
{
  Sample sa;
  sa.a =3;
  sa.b =4;
  sa.c =5;
}
```

Unbenannte Typen online lesen: https://riptutorial.com/de/cplusplus/topic/2704/unbenannte-typen

Kapitel 135: Undefiniertes Verhalten

Einführung

Was ist undefiniertes Verhalten (UB)? Gemäß dem ISO-C ++ - Standard (§ 1.3.24, N4296) ist es "Verhalten, für das diese Internationale Norm keine Anforderungen aufstellt"

Das heißt, wenn ein Programm auf UB trifft, darf es tun, was es will. Dies bedeutet oft einen Absturz, aber es kann einfach nichts tun, Dämonen aus der Nase fliegen lassen oder sogar richtig *erscheinen* zu arbeiten!

Es ist unnötig zu erwähnen, dass Sie es vermeiden sollten, Code zu schreiben, der UB aufruft.

Bemerkungen

Wenn ein Programm ein undefiniertes Verhalten enthält, setzt der C ++ - Standard keine Einschränkungen für sein Verhalten.

- Es scheint zu funktionieren, als beabsichtigte der Entwickler, aber es kann auch abstürzen oder ungewöhnliche Ergebnisse produzieren.
- Das Verhalten kann zwischen den Ausführungen desselben Programms variieren.
- Jeder Teil des Programms kann eine Fehlfunktion aufweisen, einschließlich Zeilen, die vor der Zeile stehen, die undefiniertes Verhalten enthält.
- Die Implementierung ist nicht erforderlich, um das Ergebnis undefinierten Verhaltens zu dokumentieren.

Eine Implementierung *kann* das Ergebnis einer Operation dokumentieren, die undefiniertes Verhalten gemäß dem Standard erzeugt, aber ein Programm, das von einem solchen dokumentierten Verhalten abhängt, ist nicht portierbar.

Warum undefiniertes Verhalten existiert

Intuitives, undefiniertes Verhalten wird als schlecht angesehen, da solche Fehler nicht etwa durch Ausnahmebehandler gnädig behandelt werden können.

Aber ein gewisses undefiniertes Verhalten ist eigentlich ein wesentlicher Bestandteil des Versprechens von C ++ "Sie zahlen nicht für das, was Sie nicht verwenden". Durch ein nicht definiertes Verhalten kann ein Compiler davon ausgehen, dass der Entwickler weiß, was er tut, und nicht Code einführen, um die in den obigen Beispielen hervorgehobenen Fehler zu überprüfen.

Undefiniertes Verhalten finden und vermeiden

Einige Tools können verwendet werden, um undefiniertes Verhalten während der Entwicklung zu erkennen:

- Die meisten Compiler verfügen über Warnflags, die auf einige Fälle undefinierten Verhaltens

beim Kompilieren hinweisen.
- Neuere Versionen von gcc und clang enthalten ein sogenanntes "Undefined Behavior Sanitizer" -Flag (`-fsanitize=undefined`), das zur Laufzeit nach undefiniertem Verhalten zu Leistungskosten prüft.
- `lint` -ähnlichen Werkzeuge können gründlichere undefiniertes Verhalten Analyse.

Nicht definiertes, nicht spezifiziertes und implementierungsdefiniertes Verhalten

Vom Standard C ++ 14 (ISO / IEC 14882: 2014), Abschnitt 1.9 (Programmausführung):

1. Die semantischen Beschreibungen in dieser Internationalen Norm definieren eine parametrisierte nichtdeterministische abstrakte Maschine. [SCHNITT]

2. Bestimmte Aspekte und Operationen der abstrakten Maschine werden in dieser Internationalen Norm als **implementierungsdefiniert beschrieben** (z. B. `sizeof(int)`). Diese bilden *die Parameter der abstrakten Maschine* . Jede Implementierung muss eine Dokumentation enthalten, in der ihre Merkmale und ihr Verhalten in dieser Hinsicht beschrieben werden. [SCHNITT]

3. Bestimmte andere Aspekte und Operationen der abstrakten Maschine werden in dieser Internationalen Norm als **nicht spezifiziert beschrieben** (z. B. Auswertung von Ausdrücken in einem *Neuinitialisierer,* wenn die Zuweisungsfunktion keinen Speicher belegt). Wo es möglich ist, definiert diese Internationale Norm eine Reihe zulässiger Verhaltensweisen. Diese definieren die nichtdeterministischen Aspekte der abstrakten Maschine. Eine Instanz der abstrakten Maschine kann somit mehr als eine mögliche Ausführung für ein gegebenes Programm und eine gegebene Eingabe haben.

4. Bestimmte andere Operationen werden in dieser Internationalen Norm als **undefiniert beschrieben** (oder die Auswirkungen eines Änderungsversuches eines `const` Objekts). [*Anmerkung* : Diese Internationale Norm stellt keine Anforderungen an das Verhalten von Programmen, die undefiniertes Verhalten enthalten. - *Endnote*]

Examples

Lesen oder Schreiben durch einen Nullzeiger

```
int *ptr = nullptr;
*ptr = 1; // Undefined behavior
```

Dies ist ein **undefiniertes Verhalten** , da ein Nullzeiger nicht auf ein gültiges Objekt zeigt. Daher gibt es kein Objekt bei `*ptr` in das geschrieben werden kann.

Dies verursacht zwar meistens einen Segmentierungsfehler, ist jedoch undefiniert und alles kann passieren.

Keine Rückgabeanweisung für eine Funktion mit einem nicht ungültigen Rückgabetyp

Das Auslassen der `return` Anweisung in einer Funktion, die einen nicht `void` gültigen `return` hat `void` ist **undefiniertes Verhalten** .

```
int function() {
    // Missing return statement
}

int main() {
    function(); //Undefined Behavior
}
```

Die meisten modernen Compiler geben zur Kompilierzeit eine Warnung für dieses undefinierte Verhalten aus.

Hinweis: `main` ist die einzige Ausnahme von der Regel. Wenn `main` keine `return` Anweisung hat, fügt der Compiler automatisch `return 0;` für Sie, so kann es sicher weggelassen werden.

Ändern eines String-Literal

C ++ 11

```
char *str = "hello world";
str[0] = 'H';
```

`"hello world"` ist ein Zeichenkettenliteral, dessen Modifizieren also undefiniertes Verhalten ergibt.

Die Initialisierung von `str` im obigen Beispiel wurde in C ++ 03 formal veraltet (zur Entfernung aus einer zukünftigen Version des Standards geplant). Einige Compiler vor 2003 könnten eine Warnung ausgeben (z. B. eine verdächtige Konvertierung). Nach 2003 warnen Compiler normalerweise vor einer veralteten Konvertierung.

C ++ 11

Das obige Beispiel ist ungültig und führt zu einer Compilerdiagnose in C ++ 11 und höher. Ein ähnliches Beispiel kann konstruiert werden, um undefiniertes Verhalten zu zeigen, indem die Typumwandlung explizit zugelassen wird, z.

```
char *str = const_cast<char *>("hello world");
str[0] = 'H';
```

Zugriff auf einen Out-of-Bounds-Index

Es ist nicht **definiert** , auf einen Index zuzugreifen, der für ein Array (oder einen Standard-Bibliothekscontainer, da diese alle mithilfe eines *Raw*- Arrays implementiert werden) außerhalb der Grenzen liegt:

```
int array[] = {1, 2, 3, 4, 5};
array[5] = 0;  // Undefined behavior
```

Es *darf* ein Zeiger auf das Ende des Arrays (in diesem Fall `array + 5`) verweisen. Sie können ihn nur dereferenzieren, da er kein gültiges Element ist.

```
const int *end = array + 5;  // Pointer to one past the last index
for (int *p = array; p != end; ++p)
  // Do something with `p`
```

Im Allgemeinen dürfen Sie keinen Out-of-Bounds-Zeiger erstellen. Ein Zeiger muss auf ein Element innerhalb des Arrays oder auf ein Ende nach dem Ende zeigen.

Ganzzahlige Division durch Null

```
int x = 5 / 0;    // Undefined behavior
```

Division durch `0` ist mathematisch undefiniert, und als solches macht es Sinn, dass es sich um undefiniertes Verhalten handelt.

Jedoch:

```
float x = 5.0f / 0.0f;   // x is +infinity
```

Die meisten Implementierungen implementieren IEEE-754, das die Gleitkommadivision durch Null definiert, um `NaN` (wenn der Zähler `0.0f`), `infinity` (wenn der Zähler positiv ist) oder `-infinity` (wenn der Zähler negativ ist) zurückzugeben.

Signierter Integer-Überlauf

```
int x = INT_MAX + 1;

// x can be anything -> Undefined behavior
```

> Wenn während der Auswertung eines Ausdrucks das Ergebnis nicht mathematisch definiert ist oder nicht im Bereich der für seinen Typ darstellbaren Werte liegt, ist das Verhalten undefiniert.

(C ++ 11 Standard Absatz 5/4)

Dies ist einer der unangenehmeren, da es in der Regel zu reproduzierbarem, nicht abstürzendem Verhalten führt, so dass Entwickler möglicherweise versucht sind, sich stark auf das beobachtete Verhalten zu verlassen.

Auf der anderen Seite:

```
unsigned int x = UINT_MAX + 1;
```

```
// x is 0
```

ist gut definiert seit:

> Vorzeichenlose Ganzzahlen, die als vorzeichenlos deklariert werden, müssen den Gesetzen der Arithmetik modulo `2^n` wobei `n` die Anzahl der Bits in der Wertedarstellung dieser bestimmten Ganzzahlgröße ist.

(C ++ 11 Standard Absatz 3.9.1 / 4)

Manchmal können Compiler ein undefiniertes Verhalten ausnutzen und optimieren

```
signed int x ;
if(x > x + 1)
{
    //do something
}
```

Da hier kein vorzeichenbehafteter Integer-Überlauf definiert ist, kann der Compiler davon ausgehen, dass dies möglicherweise niemals vorkommt, und kann daher den "if" -Block optimieren

Verwendung einer nicht initialisierten lokalen Variablen

```
int a;
std::cout << a; // Undefined behavior!
```

Dies führt zu **undefiniertem Verhalten** , da `a` nicht initialisiert ist.

Es wird oft fälschlicherweise behauptet, dass dies darauf zurückzuführen ist, dass der Wert "unbestimmt" ist oder "welcher Wert zuvor an diesem Speicherplatz war". Es ist jedoch der Vorgang des Zugriffs auf den Wert von `a` in dem obigen Beispiel, der undefiniertes Verhalten ergibt. In der Praxis ist das Drucken eines "Müllwerts" in diesem Fall ein häufiges Symptom. Dies ist jedoch nur eine mögliche Form von undefiniertem Verhalten.

Obwohl dies in der Praxis sehr unwahrscheinlich ist (da er auf spezifische Hardware-Unterstützung angewiesen ist), könnte der Compiler beim Programmieren des obigen Codebeispiels den Programmierer gleichfalls durch Stromschlag beschädigen. Mit einer solchen Compiler- und Hardwareunterstützung würde eine solche Reaktion auf undefiniertes Verhalten das Verständnis der durchschnittlichen (lebenden) Programmierer für die wahre Bedeutung von undefiniertem Verhalten deutlich erhöhen - was bedeutet, dass der Standard dem resultierenden Verhalten keine Beschränkungen auferlegt.

C ++ 14

Die Verwendung eines unbestimmten Werts eines `unsigned char` führt nicht zu undefiniertem Verhalten, wenn der Wert wie folgt verwendet wird:

- der zweite oder dritte Operand des ternären bedingten Operators;

- der rechte Operand des eingebauten Kommaoperators;
- der Operand einer Konvertierung in `unsigned char` ;
- der rechte Operand des Zuweisungsoperators, wenn der linke Operand ebenfalls vom Typ `unsigned char` ;
- der Initialisierer für ein Objekt `unsigned char` ;

oder wenn der Wert verworfen wird. In solchen Fällen propagiert der unbestimmte Wert gegebenenfalls einfach zum Ergebnis des Ausdrucks.

Beachten Sie, dass eine `static` Variable **immer** nullinitialisiert wird (wenn möglich):

```
static int a;
std::cout << a; // Defined behavior, 'a' is 0
```

Mehrere nicht identische Definitionen (die One-Definition-Regel)

Wenn eine Klasse, eine Enumeration, eine Inline-Funktion, eine Vorlage oder ein Member einer Vorlage über eine externe Verknüpfung verfügt und in mehreren Übersetzungseinheiten definiert ist, müssen alle Definitionen identisch sein oder das Verhalten ist gemäß der Definition der One Definition Rule (ODR) undefiniert.

`foo.h` :

```
class Foo {
  public:
    double x;
  private:
    int y;
};

Foo get_foo();
```

`foo.cpp` :

```
#include "foo.h"
Foo get_foo() { /* implementation */ }
```

`main.cpp` :

```
// I want access to the private member, so I am going to replace Foo with my own type
class Foo {
  public:
    double x;
    int y;
};
Foo get_foo(); // declare this function ourselves since we aren't including foo.h
int main() {
    Foo foo = get_foo();
    // do something with foo.y
}
```

Das obige Programm weist ein undefiniertes Verhalten auf, da es zwei Definitionen der Klasse

`::Foo` , die über eine externe Verknüpfung verfügt, in verschiedenen Übersetzungseinheiten. Die beiden Definitionen sind jedoch nicht identisch. Im Gegensatz zur Neudefinition einer Klasse innerhalb *derselben* Übersetzungseinheit muss dieses Problem nicht vom Compiler diagnostiziert werden.

Falsche Paarung von Speicherzuweisung und Freigabe

Ein Objekt kann nur dann `delete` wenn es von `new` zugewiesen wurde und kein Array ist. Wenn das zu `delete` Argument nicht von `new` oder ein Array ist, ist das Verhalten undefiniert.

Ein Objekt kann nur durch `delete[]` freigegeben werden, wenn es von `new` zugewiesen wurde und ein Array ist. Wenn das zu `delete[]` Argument `delete[]` nicht von `new` oder kein Array ist, ist das Verhalten undefiniert.

Wenn das Argument `free` nicht von `malloc` , ist das Verhalten undefiniert.

```
int* p1 = new int;
delete p1;      // correct
// delete[] p1; // undefined
// free(p1);    // undefined

int* p2 = new int[10];
delete[] p2;    // correct
// delete p2;   // undefined
// free(p2);    // undefined

int* p3 = static_cast<int*>(malloc(sizeof(int)));
free(p3);       // correct
// delete p3;   // undefined
// delete[] p3; // undefined
```

Solche Probleme können vermieden werden, indem Sie `malloc` und `free` in C ++ - Programmen vollständig vermeiden, die intelligenten Zeiger der Standardbibliothek gegenüber raw `new` und `delete` und `std::vector` und `std::string` gegenüber raw `new` und `delete[]` vorziehen.

Zugriff auf ein Objekt als falscher Typ

In den meisten Fällen ist es illegal, auf ein Objekt eines Typs zuzugreifen, als wäre es ein anderer Typ (ohne Berücksichtigung von cv-qualifiers). Beispiel:

```
float x = 42;
int y = reinterpret_cast<int&>(x);
```

Das Ergebnis ist undefiniertes Verhalten.

Es gibt einige Ausnahmen von dieser *strengen Aliasing*- Regel:

- Auf ein Objekt des Klassentyps kann zugegriffen werden, als wäre es ein Typ, der eine Basisklasse des tatsächlichen Klassentyps ist.
- Auf einen beliebigen Typ kann als `char` oder als `unsigned char` zugegriffen werden. Das Gegenteil ist jedoch nicht der Fall: Auf ein Zeichenarray kann nicht wie bei einem beliebigen

Typ zugegriffen werden.
- Auf einen vorzeichenbehafteten Integer-Typ kann als entsprechender vorzeichenloser Typ zugegriffen werden und *umgekehrt* .

Eine verwandte Regel besagt, dass beim Aufruf einer nicht statischen Memberfunktion für ein Objekt, das nicht denselben Typ wie die definierende Klasse der Funktion oder eine abgeleitete Klasse hat, ein undefiniertes Verhalten auftritt. Dies gilt auch dann, wenn die Funktion nicht auf das Objekt zugreift.

```
struct Base {
};
struct Derived : Base {
    void f() {}
};
struct Unrelated {};
Unrelated u;
Derived& r1 = reinterpret_cast<Derived&>(u); // ok
r1.f();                                      // UB
Base b;
Derived& r2 = reinterpret_cast<Derived&>(b); // ok
r2.f();                                      // UB
```

Fließpunktüberlauf

Wenn eine arithmetische Operation, die einen Gleitkommatyp ergibt, einen Wert ergibt, der nicht im Bereich der darstellbaren Werte des Ergebnistyps liegt, ist das Verhalten gemäß dem C ++ - Standard undefiniert, kann aber durch andere Standards definiert werden, denen die Maschine möglicherweise entspricht. wie IEEE 754.

```
float x = 1.0;
for (int i = 0; i < 10000; i++) {
    x *= 10.0; // will probably overflow eventually; undefined behavior
}
```

(Reine) virtuelle Member vom Konstruktor oder Destruktor aufrufen

Der Standard (10.4) besagt:

> Elementfunktionen können von einem Konstruktor (oder Destruktor) einer abstrakten Klasse aufgerufen werden. Der Effekt, einen virtuellen Aufruf (10.3) direkt oder indirekt für eine reine virtuelle Funktion für das Objekt auszuführen, das von einem solchen Konstruktor (oder Destruktor) erstellt (oder zerstört) wird, ist nicht definiert.

Im Allgemeinen schlagen einige C ++ - Behörden, z. B. Scott Meyers, vor , virtuelle Funktionen (auch nicht reine Funktionen) nie von Konstruktoren und Konstruktoren aufzurufen.

Betrachten Sie das folgende Beispiel, das vom obigen Link geändert wurde:

```
class transaction
{
public:
```

```
    transaction(){ log_it(); }
    virtual void log_it() const = 0;
};

class sell_transaction : public transaction
{
public:
    virtual void log_it() const { /* Do something */ }
};
```

Angenommen, wir erstellen ein `sell_transaction` Objekt:

```
sell_transaction s;
```

Dies ruft implizit den Konstruktor von `sell_transaction` , der zuerst den Konstruktor von `transaction` aufruft. Wenn der Konstruktor von `transaction` aufgerufen wird, ist das Objekt jedoch noch nicht vom Typ `sell_transaction` , sondern nur vom Typ `transaction` .

Folglich führt der Aufruf von `transaction::transaction()` zu `log_it` nicht zu dem, was vielleicht intuitiv erscheint - nämlich `sell_transaction::log_it` .

- Wenn `log_it` wie in diesem Beispiel rein virtuell ist, ist das Verhalten undefiniert.
- Wenn `log_it` nicht rein virtuell ist, wird `transaction::log_it` aufgerufen.

Löschen eines abgeleiteten Objekts über einen Zeiger auf eine Basisklasse, die keinen virtuellen Destruktor besitzt.

```
class base { };
class derived: public base { };

int main() {
    base* p = new derived();
    delete p; // The is undefined behavior!
}
```

In Abschnitt [expr.delete] §5.3.5 / 3 sagt der Standard, dass wenn `delete` für ein Objekt aufgerufen wird, dessen statischer Typ keinen `virtual` Destruktor hat:

> Wenn sich der statische Typ des zu löschenden Objekts von seinem dynamischen Typ unterscheidet, muss der statische Typ eine Basisklasse des dynamischen Typs des zu löschenden Objekts sein und der statische Typ muss einen virtuellen Destruktor haben oder das Verhalten ist undefiniert.

Dies ist unabhängig von der Frage der Fall, ob die abgeleitete Klasse Datenelemente zur Basisklasse hinzugefügt hat.

Zugriff auf eine baumelnde Referenz

Es ist illegal, auf einen Verweis auf ein Objekt zuzugreifen, das den Gültigkeitsbereich verlassen hat oder auf andere Weise zerstört wurde. Es wird gesagt, dass eine solche Referenz *baumelt,* da

sie sich nicht mehr auf ein gültiges Objekt bezieht.

```
#include <iostream>
int& getX() {
    int x = 42;
    return x;
}
int main() {
    int& r = getX();
    std::cout << r << "\n";
}
```

In diesem Beispiel `getX` die lokale Variable `x` Gültigkeitsbereich, wenn `getX` zurückgegeben wird. (Beachten Sie, dass die *Lebensdauerverlängerung* die Lebensdauer einer lokalen Variablen nicht über den Gültigkeitsbereich des Blocks hinaus verlängern kann, in dem sie definiert ist.) Daher ist `r` eine schwankende Referenz. Dieses Programm hat das Verhalten nicht definiert, obwohl es zu arbeiten und drucken erscheint `42` in einigen Fällen.

Erweiterung des "std" oder "posix" Namespaces

Der Standard (17.6.4.2.1 / 1) verbietet im Allgemeinen die Erweiterung des `std` Namespaces:

> Das Verhalten eines C ++ - Programms ist undefiniert, wenn es Deklarationen oder Definitionen zum Namespace std oder einem Namespace innerhalb des Namespace std hinzufügt, sofern nichts anderes angegeben ist.

Gleiches gilt für `posix` (17.6.4.2.2 / 1):

> Das Verhalten eines C ++ - Programms ist undefiniert, wenn es Deklarationen oder Definitionen zum Namespace posix oder einem Namespace innerhalb des Namespace posix hinzufügt, sofern nichts anderes angegeben ist.

Folgendes berücksichtigen:

```
#include <algorithm>

namespace std
{
    int foo(){}
}
```

Nichts im Standard verbietet den `algorithm` (oder einen der darin enthaltenen Header), die dieselbe Definition definieren. Daher würde dieser Code gegen die One-Definitions-Regel verstoßen.

Das ist also generell verboten. Es sind jedoch bestimmte Ausnahmen zulässig . Am nützlichsten ist es vielleicht, Spezialisierungen für benutzerdefinierte Typen hinzuzufügen. Nehmen Sie beispielsweise an, Ihr Code hat dies

```
class foo
{
```

```
    // Stuff
};
```

Dann ist das Folgende in Ordnung

```
namespace std
{
    template<>
    struct hash<foo>
    {
    public:
        size_t operator()(const foo &f) const;
    };
}
```

Überlauf während der Konvertierung in oder vom Gleitkommatyp

Wenn während der Umwandlung von:

- einen ganzzahligen Typ zu einem Gleitkommatyp,
- einem Gleitkommatyp zu einem Ganzzahlentyp oder
- einem Gleitkommatyp zu einem kürzeren Gleitkommatyp,

Der Quellwert liegt außerhalb des Wertebereichs, der im Zieltyp dargestellt werden kann. Das Ergebnis ist undefiniertes Verhalten. Beispiel:

```
double x = 1e100;
int y = x; // int probably cannot hold numbers that large, so this is UB
```

Ungültige statische Umwandlung von Basis zu abgeleiteten Elementen

Wenn mit `static_cast` ein Zeiger (bzw. eine Referenz) zur Basisklasse in einen Zeiger (bzw. eine Referenz) für eine abgeleitete Klasse konvertiert wird, zeigt der Operand jedoch nicht auf ein Objekt des abgeleiteten Klassentyps (Verhalten) ist nicht definiert. Siehe Konvertierung von Basis zu abgeleiteter Klasse .

Funktionsaufruf durch nicht übereinstimmenden Funktionszeigertyp

Um eine Funktion über einen Funktionszeiger aufzurufen, muss der Typ des Funktionszeigers genau dem Typ der Funktion entsprechen. Ansonsten ist das Verhalten undefiniert. Beispiel:

```
int f();
void (*p)() = reinterpret_cast<void(*)()>(f);
p(); // undefined
```

Ein const-Objekt ändern

Jeder Versuch, ein `const` Objekt zu ändern, führt zu undefiniertem Verhalten. Dies gilt für `const` Variablen, Member von `const` Objekten und für `const` deklarierte Klassenmitglieder. (Ein `mutable`

Member eines `const` Objekts ist jedoch nicht `const` .)

Ein solcher Versuch kann mit `const_cast` :

```
const int x = 123;
const_cast<int&>(x) = 456;
std::cout << x << '\n';
```

Ein Compiler wird inline in der Regel den Wert eines `const int` - Objekt, ist es so möglich , dass dieser Code kompiliert und druckt `123` . Compiler können die Werte von `const` Objekten auch in den Nur-Lese-Speicher legen, sodass ein Segmentierungsfehler auftreten kann. In jedem Fall ist das Verhalten undefiniert und das Programm kann irgendetwas tun.

Das folgende Programm verbirgt einen weitaus subtileren Fehler:

```
#include <iostream>

class Foo* instance;

class Foo {
  public:
    int get_x() const { return m_x; }
    void set_x(int x) { m_x = x; }
  private:
    Foo(int x, Foo*& this_ref): m_x(x) {
        this_ref = this;
    }
    int m_x;
    friend const Foo& getFoo();
};

const Foo& getFoo() {
    static const Foo foo(123, instance);
    return foo;
}

void do_evil(int x) {
    instance->set_x(x);
}

int main() {
    const Foo& foo = getFoo();
    do_evil(456);
    std::cout << foo.get_x() << '\n';
}
```

In diesem Code erstellt `getFoo` ein Singleton vom Typ `const Foo` und sein Member `m_x` wird mit `123` initialisiert. Dann wird `do_evil` aufgerufen und der Wert von `foo.m_x` wird offensichtlich in 456 geändert. Was ist schief gelaufen?

Trotz seines Namens tut `do_evil` nichts besonders Böses; Alles, was es tut, ist ein Setter mit einem `Foo*` anzurufen. Dieser Zeiger zeigt jedoch auf ein `const Foo` Objekt, obwohl `const_cast` nicht verwendet wurde. Dieser Zeiger wurde durch den Konstruktor von `Foo` . Ein `const` - Objekt wird nicht `const` , bis seine Initialisierung abgeschlossen ist, so `this` hat Typ `Foo*` , nicht `const Foo*` , im Konstruktor.

Daher tritt undefiniertes Verhalten auf, obwohl in diesem Programm keine offensichtlich gefährlichen Konstrukte vorhanden sind.

Zugriff auf nicht vorhandenes Mitglied durch Zeiger auf Mitglied

Wenn auf ein nicht statisches Member eines Objekts über einen Zeiger auf Member zugegriffen wird, ist das Verhalten undefiniert, wenn das Objekt tatsächlich nicht das durch den Pointer angegebene Member enthält. (Ein solcher Zeiger auf member kann über `static_cast` abgerufen werden.)

```
struct Base { int x; };
struct Derived : Base { int y; };
int Derived::*pdy = &Derived::y;
int Base::*pby = static_cast<int Base::*>(pdy);

Base* b1 = new Derived;
b1->*pby = 42; // ok; sets y in Derived object to 42
Base* b2 = new Base;
b2->*pby = 42; // undefined; there is no y member in Base
```

Ungültige Umwandlung von Basis zu Basis für Zeiger auf Mitglieder

Wenn mit `static_cast TD::*` in `TB::*` konvertiert wird, muss das Member, auf das verwiesen wird, zu einer Klasse gehören, die eine Basisklasse oder eine abgeleitete Klasse von `B` . Ansonsten ist das Verhalten undefiniert. Weitere Informationen finden Sie unter Abgeleitete in Basisumwandlung für Zeiger auf Mitglieder

Ungültige Zeigerarithmetik

Die folgenden Verwendungen von Zeigerarithmetik verursachen undefiniertes Verhalten:

- Addition oder Subtraktion einer Ganzzahl, wenn das Ergebnis nicht zu demselben Array-Objekt gehört wie der Zeigeroperand. (Hier gilt das Element eins nach dem Ende als zum Array gehörig.)

  ```
  int a[10];
  int* p1 = &a[5];
  int* p2 = p1 + 4; // ok; p2 points to a[9]
  int* p3 = p1 + 5; // ok; p2 points to one past the end of a
  int* p4 = p1 + 6; // UB
  int* p5 = p1 - 5; // ok; p2 points to a[0]
  int* p6 = p1 - 6; // UB
  int* p7 = p3 - 5; // ok; p7 points to a[5]
  ```

- Subtraktion von zwei Zeigern, wenn beide nicht zum selben Array-Objekt gehören. (Wiederum wird das Element um eins nach dem Ende als zu dem Array gehörig betrachtet.) Die Ausnahme ist, dass zwei Nullzeiger subtrahiert werden können, was 0 ergibt.

  ```
  int a[10];
  int b[10];
  ```

```
int *p1 = &a[8], *p2 = &a[3];
int d1 = p1 - p2; // yields 5
int *p3 = p1 + 2; // ok; p3 points to one past the end of a
int d2 = p3 - p2; // yields 7
int *p4 = &b[0];
int d3 = p4 - p1; // UB
```

- Subtraktion von zwei Zeigern, wenn das Ergebnis `std::ptrdiff_t` .
- Jede Zeigerarithmetik, bei der der Pointee-Typ eines Operanden nicht mit dem dynamischen Typ des Objekts übereinstimmt, auf das er zeigt (Ignorieren der cv-Qualifikation) Gemäß dem Standard "kann insbesondere ein Zeiger auf eine Basisklasse nicht für die Zeigerarithmetik verwendet werden, wenn das Array Objekte eines abgeleiteten Klassentyps enthält."

```
struct Base { int x; };
struct Derived : Base { int y; };
Derived a[10];
Base* p1 = &a[1];           // ok
Base* p2 = p1 + 1;          // UB; p1 points to Derived
Base* p3 = p1 - 1;          // likewise
Base* p4 = &a[2];           // ok
auto p5 = p4 - p1;          // UB; p4 and p1 point to Derived
const Derived* p6 = &a[1];
const Derived* p7 = p6 + 1; // ok; cv-qualifiers don't matter
```

Verschiebung um eine ungültige Anzahl von Positionen

Für den eingebauten Verschiebungsoperator muss der rechte Operand nicht negativ sein und ist strikt unter der Bitbreite des beförderten linken Operanden. Ansonsten ist das Verhalten undefiniert.

```
const int a = 42;
const int b = a << -1; // UB
const int c = a << 0;  // ok
const int d = a << 32; // UB if int is 32 bits or less
const int e = a >> 32; // also UB if int is 32 bits or less
const signed char f = 'x';
const int g = f << 10; // ok even if signed char is 10 bits or less;
                       // int must be at least 16 bits
```

Rückkehr von einer [[noreturn]] - Funktion

C ++ 11

Beispiel aus dem Standard, [dcl.attr.noreturn]:

```
[[ noreturn ]] void f() {
  throw "error"; // OK
}
[[ noreturn ]] void q(int i) { // behavior is undefined if called with an argument <= 0
  if (i > 0)
    throw "positive";
```

```
}
```

Zerstörung eines bereits zerstörten Objekts

In diesem Beispiel wird ein Destruktor explizit für ein Objekt aufgerufen, das später automatisch gelöscht wird.

```
struct S {
    ~S() { std::cout << "destroying S\n"; }
};
int main() {
    S s;
    s.~S();
} // UB: s destroyed a second time here
```

Ein ähnliches Problem tritt auf, wenn ein `std::unique_ptr<T>` auf ein `T` mit automatischer oder statischer Speicherdauer verweist.

```
void f(std::unique_ptr<S> p);
int main() {
    S s;
    std::unique_ptr<S> p(&s);
    f(std::move(p)); // s destroyed upon return from f
}                    // UB: s destroyed
```

Eine andere Möglichkeit, ein Objekt zweimal zu zerstören, besteht darin, dass zwei `shared_ptr` s das Objekt verwalten, ohne dass der Besitz miteinander geteilt wird.

```
void f(std::shared_ptr<S> p1, std::shared_ptr<S> p2);
int main() {
    S* p = new S;
    // I want to pass the same object twice...
    std::shared_ptr<S> sp1(p);
    std::shared_ptr<S> sp2(p);
    f(sp1, sp2);
} // UB: both sp1 and sp2 will destroy s separately
// NB: this is correct:
// std::shared_ptr<S> sp(p);
// f(sp, sp);
```

Unendliche Vorlagenrekursion

Beispiel aus dem Standard, [temp.inst] / 17:

```
template<class T> class X {
    X<T>* p; // OK
    X<T*> a; // implicit generation of X<T> requires
             // the implicit instantiation of X<T*> which requires
             // the implicit instantiation of X<T**> which ...
};
```

Undefiniertes Verhalten online lesen: https://riptutorial.com/de/cplusplus/topic/1812/undefiniertes-

verhalten

Kapitel 136: Variablendeklarationsschlüsselwörter

Examples

const

Einen Typbezeichner; Bei Anwendung auf einen Typ wird die const-qualifizierte Version des Typs erstellt. Weitere Informationen zur Bedeutung von `const` Sie unter const-Schlüsselwort .

```
const int x = 123;
x = 456;    // error
int& r = x; // error

struct S {
    void f();
    void g() const;
};
const S s;
s.f(); // error
s.g(); // OK
```

decltype

C ++ 11

Gibt den Typ seines Operanden aus, der nicht ausgewertet wird.

- Wenn der Operand `e` ein Name ohne zusätzliche Klammern ist, dann ist `decltype(e)` der *deklarierte Typ* von `e` .

  ```
  int x = 42;
  std::vector<decltype(x)> v(100, x); // v is a vector<int>
  ```

- Wenn der Operand `e` ein Klassenmitgliedszugriff ohne zusätzliche Klammern ist, dann ist `decltype(e)` der *deklarierte Typ* des Mitglieds, auf das zugegriffen wird.

  ```
  struct S {
      int x = 42;
  };
  const S s;
  decltype(s.x) y; // y has type int, even though s.x is const
  ```

- In allen anderen Fällen liefert `decltype(e)` sowohl den Typ als auch die Wertkategorie des Ausdrucks `e` wie folgt:

  - Wenn `e` ein lWert vom Typ `T` , dann ist `decltype(e) T&` .
  - Wenn `e` ein x-Wert vom Typ `T` , ist `decltype(e) T&&` .

- Wenn `e` ein Wert vom Typ `T` , dann ist der Typ `decltype(e) T`

Dies schließt den Fall mit äußeren Klammern ein.

```
int f() { return 42; }
int& g() { static int x = 42; return x; }
int x = 42;
decltype(f()) a = f(); // a has type int
decltype(g()) b = g(); // b has type int&
decltype((x)) c = x;   // c has type int&, since x is an lvalue
```

C ++ 14

Das spezielle Formular `decltype(auto)` leitet den Typ einer Variablen von ihrem Initialisierer oder den Rückgabetyp einer Funktion aus den `return` Anweisungen in ihrer Definition ab, wobei sie die `decltype` von `decltype` und nicht die von `auto` .

```
const int x = 123;
auto y = x;           // y has type int
decltype(auto) z = x; // z has type const int, the declared type of x
```

unterzeichnet

Ein Schlüsselwort, das Teil bestimmter Integer-Typnamen ist.

- Wenn es alleine verwendet wird, ist `int` impliziert, so dass `signed` , `signed int` und `int` vom gleichen Typ sind.
- In Kombination mit `char` ergibt sich der Typ `signed char` , der sich von `char` , auch wenn `char` signiert ist. `signed char` Zeichen umfasst mindestens -127 bis +127.
- In Kombination mit `short` , `long` oder `long long` ist es überflüssig, da diese Typen bereits signiert sind.
- `signed` kann nicht mit `bool` , `wchar_t` , `char16_t` oder `char32_t` .

Beispiel:

```
signed char celsius_temperature;
std::cin >> celsius_temperature;
if (celsius_temperature < -35) {
    std::cout << "cold day, eh?\n";
}
```

ohne Vorzeichen

Ein Typbezeichner, der die vorzeichenlose Version eines Integer-Typs anfordert.

- Wenn es alleine verwendet wird, ist `int` impliziert, also ist `unsigned` Typ derselbe Typ wie `unsigned int` .
- Der Typ `unsigned char` unterscheidet sich vom Typ `char` , auch wenn `char` unsigniert ist. Es kann ganze Zahlen bis zu 255 enthalten.
- `unsigned` kann auch mit `short` , `long` oder `long long` kombiniert werden. Es kann nicht mit `bool`

, `wchar_t` , `char16_t` oder `char32_t` .

Beispiel:

```
char invert_case_table[256] = { ..., 'a', 'b', 'c', ..., 'A', 'B', 'C', ... };
char invert_case(char c) {
    unsigned char index = c;
    return invert_case_table[index];
    // note: returning invert_case_table[c] directly does the
    // wrong thing on implementations where char is a signed type
}
```

flüchtig

Eine Typqualifikation; Bei Anwendung auf einen Typ wird die volatile-qualifizierte Version des Typs erstellt. Flüchtige Qualifikation spielt die gleiche Rolle wie `const` Qualifikation in dem Typsystem, aber `volatile` nicht daran hindert , Objekte, verändert; stattdessen zwingt der Compiler alle Zugriffe auf solche Objekte als Nebeneffekte.

Wenn im Beispiel unten, `memory_mapped_port` nicht flüchtig, so könnte der Compiler die Funktion optimieren , so dass es nur die letzte Schreib führt, was falsch wäre , wenn `sizeof(int)` ist größer als 1. Die `volatile` Qualifikation Kräfte , um alles zu behandeln `sizeof(int)` schreibt als verschiedene Nebenwirkungen und führt sie daher alle (in Reihenfolge) aus.

```
extern volatile char memory_mapped_port;
void write_to_device(int x) {
    const char* p = reinterpret_cast<const char*>(&x);
    for (int i = 0; i < sizeof(int); i++) {
        memory_mapped_port = p[i];
    }
}
```

Variablendeklarationsschlüsselwörter online lesen:
https://riptutorial.com/de/cplusplus/topic/7840/variablendeklarationsschlusselworter

Kapitel 137: veränderbares Schlüsselwort

Examples

Nicht statischer Klassenmitgliedsmodifizierer

`mutable` Modifizierer wird in diesem Zusammenhang verwendet, um anzuzeigen, dass ein Datenfeld eines const-Objekts geändert werden kann, ohne den von außen sichtbaren Zustand des Objekts zu beeinflussen.

Wenn Sie über das Zwischenspeichern eines Ergebnisses einer kostspieligen Berechnung nachdenken, sollten Sie dieses Schlüsselwort wahrscheinlich verwenden.

Wenn Sie ein Sperrdatenfeld (z. B. `std::unique_lock`) haben, das in einer const-Methode gesperrt und nicht gesperrt ist, können Sie dieses Schlüsselwort auch verwenden.

Sie sollten dieses Schlüsselwort nicht verwenden, um die logische Konstanz eines Objekts zu unterbrechen.

Beispiel mit Caching:

```
class pi_calculator {
 public:
     double get_pi() const {
         if (pi_calculated) {
             return pi;
         } else {
             double new_pi = 0;
             for (int i = 0; i < 1000000000; ++i) {
                 // some calculation to refine new_pi
             }
             // note: if pi and pi_calculated were not mutable, we would get an error from a
compiler
             // because in a const method we can not change a non-mutable field
             pi = new_pi;
             pi_calculated = true;
             return pi;
         }
     }
 private:
     mutable bool pi_calculated = false;
     mutable double pi = 0;
};
```

veränderliche Lambdas

Der implizite `operator()` eines Lambda ist standardmäßig `const` . Diese verbietet notleidender `const` auf der Lambda - Operationen. Um modifizierende Elemente zuzulassen, kann ein Lambda als `mutable` markiert werden, wodurch der implizite `operator()` nicht- `const` :

```
int a = 0;

auto bad_counter = [a] {
    return a++;   // error: operator() is const
                  // cannot modify members
};

auto good_counter = [a]() mutable {
    return a++;  // OK
}

good_counter(); // 0
good_counter(); // 1
good_counter(); // 2
```

veränderbares Schlüsselwort online lesen:
https://riptutorial.com/de/cplusplus/topic/2705/veranderbares-schlusselwort

Kapitel 138: Verweise

Examples

Referenz definieren

Referenzen verhalten sich ähnlich, aber nicht ganz wie const-Zeiger. Eine Referenz wird definiert, indem ein kaufmännisches Und `&` an einen Typnamen angehängt wird.

```
int i = 10;
int &refi = i;
```

Hier ist `refi` eine Referenz, die an `i` gebunden ist.
References abstrahieren die Semantik von Zeigern und wirken wie ein Alias für das zugrunde liegende Objekt:

```
refi = 20; // i = 20;
```

Sie können auch mehrere Referenzen in einer einzigen Definition definieren:

```
int i = 10, j = 20;
int &refi = i, &refj = j;

// Common pitfall :
// int& refi = i, k = j;
// refi will be of type int&.
// though, k will be of type int, not int&!
```

Referenzen **müssen** zum Zeitpunkt der Definition korrekt initialisiert werden und können danach nicht mehr geändert werden. Der folgende Code verursacht einen Kompilierungsfehler:

```
int &i; // error: declaration of reference variable 'i' requires an initializer
```

`nullptr` Gegensatz zu Zeigern können Sie auch keinen direkten Verweis auf `nullptr` binden:

```
int *const ptri = nullptr;
int &refi = nullptr; // error: non-const lvalue reference to type 'int' cannot bind to a
temporary of type 'nullptr_t'
```

C ++ - Referenzen sind Alias für vorhandene Variablen

Eine Referenz in C ++ ist nur ein `Alias` oder ein anderer Name einer Variablen. Genau wie die meisten von uns können wir sie mit unserem Passnamen und Nickname beziehen.

Referenzen existieren nicht wörtlich und sie belegen keinen Speicher. Wenn wir die Adresse der Referenzvariablen ausgeben, wird dieselbe Adresse ausgegeben wie die Variable, auf die sie verweist.

```
int main() {
    int i = 10;
    int &j = i;

    cout<<&i<<endl;
    cout<<&b<<endl;
    return 0;
}
```

Im obigen Beispiel drucken beide `cout` Adressen dieselbe Adresse. Die Situation wird dieselbe sein, wenn wir eine Variable als Referenz in einer Funktion verwenden

```
void func (int &fParam ) {
   cout<<"Address inside function => "<<fParam<<endl;
}

int main() {
    int i = 10;
    cout<<"Address inside Main => "<<&i<<endl;

    func(i);

    return 0;
}
```

Auch in diesem Beispiel drucken beide `cout` dieselbe Adresse.

Da wir inzwischen wissen, dass `C++ References` nur ein Alias sind und ein Alias erstellt werden muss, müssen wir etwas haben, auf das der Alias verweisen kann.

Das ist der genaue Grund, warum die Anweisung einen Compiler-Fehler auslöst

```
int &i;
```

Denn der Alias bezieht sich nicht auf irgendetwas.

Verweise online lesen: https://riptutorial.com/de/cplusplus/topic/1548/verweise

Kapitel 139: Verwenden von std :: unordered_map

Einführung

std :: unordered_map ist nur ein assoziativer Container. Es funktioniert auf Schlüsseln und deren Karten. Key, wie der Name sagt, hilft, Eindeutigkeit in der Karte zu haben. Der zugeordnete Wert ist nur ein Inhalt, der dem Schlüssel zugeordnet ist. Die Datentypen dieses Schlüssels und dieser Zuordnung können jeder vordefinierte oder benutzerdefinierte Datentyp sein.

Bemerkungen

Wie der Name sagt, werden die Elemente in der ungeordneten Karte nicht in sortierter Reihenfolge gespeichert. Sie werden entsprechend ihren Hashwerten gespeichert. Daher hat die Verwendung einer ungeordneten Karte viele Vorteile, da zum Durchsuchen eines Elements nur O (1) erforderlich ist. Es ist auch schneller als andere Kartencontainer. Das Beispiel zeigt auch, dass es sehr einfach zu implementieren ist, da der Operator ([]) uns hilft, direkt auf den zugeordneten Wert zuzugreifen.

Examples

Erklärung und Verwendung

Wie bereits erwähnt, können Sie eine ungeordnete Karte eines beliebigen Typs deklarieren. Wir haben eine ungeordnete Map mit dem String- und Integer-Typ.

```
unordered_map<string, int> first; //declaration of the map
first["One"] = 1; // [] operator used to insert the value
first["Two"] = 2;
first["Three"] = 3;
first["Four"] = 4;
first["Five"] = 5;

pair <string,int> bar = make_pair("Nine", 9); //make a pair of same type
first.insert(bar); //can also use insert to feed the values
```

Einige Grundfunktionen

```
unordered_map<data_type, data_type> variable_name; //declaration

variable_name[key_value] = mapped_value; //inserting values

variable_name.find(key_value); //returns iterator to the key value

variable_name.begin(); // iterator to the first element
```

```
variable_name.end(); // iterator to the last + 1 element
```

Verwenden von std :: unordered_map online lesen:
https://riptutorial.com/de/cplusplus/topic/10540/verwenden-von-std----unordered-map

Kapitel 140: Virtuelle Elementfunktionen

Syntax

- virtuelle Leere f ();
- virtuelle Leere g () = 0;
- // C ++ 11 oder höher:
    - virtuelle Leere h () überschreiben;
    - void i () überschreiben;
    - virtuelle Leere j () final;
    - ungültig k () final;

Bemerkungen

- Nur nicht statische Member-Funktionen, die keine Vorlagen sind, können `virtual` .
- Wenn Sie C ++ 11 oder höher verwenden, wird empfohlen, beim Überschreiben einer virtuellen Memberfunktion von einer Basisklasse die `override` zu verwenden.
- Polymorphe Basisklassen verfügen häufig über virtuelle Destruktoren, damit ein abgeleitetes Objekt über einen Zeiger auf die Basisklasse gelöscht werden kann . Wenn der Destruktor nicht virtuell war, führt eine solche Operation zu undefiniertem Verhalten [expr.delete] §5.3.5 / 3 .

Examples

Verwenden der Überschreibung mit virtual in C ++ 11 und höher

Das `override` hat in C ++ 11 eine besondere Bedeutung, wenn es am Ende der Funktionssignatur angehängt wird. Dies bedeutet, dass eine Funktion ist

- Überschreiben der Funktion in der Basisklasse &
- Die Basisklassenfunktion ist `virtual`

Dieser Spezifizierer hat keine `run time` da er hauptsächlich als Hinweis für Compiler gedacht ist

Das folgende Beispiel zeigt die Änderung des Verhaltens bei Verwendung ohne Überschreiben.

Ohne `override` :

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; }
};
```

```
struct Y : X {
    // Y::f() will not override X::f() because it has a different signature,
    // but the compiler will accept the code (and silently ignore Y::f()).
    virtual void f(int a) { std::cout << a << "\n"; }
};
```

Mit `override` :

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
    // The compiler will alert you to the fact that Y::f() does not
    // actually override anything.
    virtual void f(int a) override { std::cout << a << "\n"; }
};
```

Beachten Sie, dass `override` kein Schlüsselwort ist, sondern ein spezieller Bezeichner, der nur in Funktionssignaturen vorkommen kann. In allen anderen Kontexten kann `override` noch als Bezeichner verwendet werden:

```
void foo() {
    int override = 1; // OK.
    int virtual = 2;  // Compilation error: keywords can't be used as identifiers.
}
```

Virtuelle vs. nicht virtuelle Memberfunktionen

Mit virtuellen Memberfunktionen:

```
#include <iostream>

struct X {
    virtual void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
    // Specifying virtual again here is optional
    // because it can be inferred from X::f().
    virtual void f() { std::cout << "Y::f()\n"; }
};

void call(X& a) {
    a.f();
}

int main() {
    X x;
    Y y;
    call(x); // outputs "X::f()"
    call(y); // outputs "Y::f()"
}
```

Ohne virtuelle Memberfunktionen:

```
#include <iostream>

struct X {
   void f() { std::cout << "X::f()\n"; }
};

struct Y : X {
   void f() { std::cout << "Y::f()\n"; }
};

void call(X& a) {
    a.f();
}

int main() {
    X x;
    Y y;
    call(x); // outputs "X::f()"
    call(y); // outputs "X::f()"
}
```

Letzte virtuelle Funktionen

In C ++ 11 wurde der `final` Bezeichner eingeführt, der das Überschreiben von Methoden verbietet, wenn sie in der Methodensignatur erscheinen:

```
class Base {
public:
    virtual void foo() {
        std::cout << "Base::Foo\n";
    }
};

class Derived1 : public Base {
public:
    // Overriding Base::foo
    void foo() final {
        std::cout << "Derived1::Foo\n";
    }
};

class Derived2 : public Derived1 {
public:
    // Compilation error: cannot override final method
    virtual void foo() {
        std::cout << "Derived2::Foo\n";
    }
};
```

Der Spezifizierer `final` kann nur mit der "virtuellen" Member-Funktion verwendet werden und kann nicht auf nicht-virtuelle Member-Funktionen angewendet werden

Wie `final` gibt es auch einen Spezifizierer, der das Überschreiben von `virtual` Funktionen in der abgeleiteten Klasse verhindert.

Die Spezifizierer `override` und `final` können kombiniert werden, um die gewünschte Wirkung zu erzielen:

```
class Derived1 : public Base {
public:
    void foo() final override {
        std::cout << "Derived1::Foo\n";
    }
};
```

Verhalten virtueller Funktionen in Konstruktoren und Destruktoren

Das Verhalten von virtuellen Funktionen in Konstruktoren und Destruktoren ist beim ersten Auftreten oft verwirrend.

```
#include <iostream>
using namespace std;

class base {
public:
    base() { f("base constructor"); }
    ~base() { f("base destructor"); }

    virtual const char* v() { return "base::v()"; }

    void f(const char* caller) {
        cout << "When called from " << caller << ", "  << v() << " gets called.\n";
    }
};

class derived : public base {
public:
    derived() { f("derived constructor"); }
    ~derived() { f("derived destructor"); }

    const char* v() override { return "derived::v()"; }

};

int main() {
     derived d;
}
```

Ausgabe:

Beim Aufruf vom Basiskonstruktor wird base :: v () aufgerufen.
Beim Aufruf aus dem abgeleiteten Konstruktor wird die abgeleitete :: v () - Anweisung aufgerufen.
Wenn der abgeleitete Destruktor aufgerufen wird, wird der abgeleitete :: v () aufgerufen.
Beim Aufruf vom Basis-Destruktor wird base :: v () aufgerufen.

Der Grund dafür ist, dass die abgeleitete Klasse zusätzliche Member definieren kann, die noch nicht initialisiert sind (im Konstruktorfall) oder bereits zerstört wurden (im Destruktorfall). Daher

wird der *dynamische* Typ von `*this` beim Konstruieren und Zerstören von C ++ - Objekten als Klasse des Konstruktors oder Destruktors und nicht als eine abgeleitete Klasse betrachtet.

Beispiel:

```
#include <iostream>
#include <memory>

using namespace std;
class base {
public:
    base()
    {
        std::cout << "foo is " << foo() << std::endl;
    }
    virtual int foo() { return 42; }
};

class derived : public base {
    unique_ptr<int> ptr_;
public:
    derived(int i) : ptr_(new int(i*i)) { }
    // The following cannot be called before derived::derived due to how C++ behaves,
    // if it was possible... Kaboom!
    int foo() override   { return *ptr_; }
};

int main() {
    derived d(4);
}
```

Reine virtuelle Funktionen

Wir können auch angeben, dass eine `virtual` Funktion *rein virtuell* (abstrakt) ist, indem Sie an die Deklaration `= 0` anhängen. Klassen mit einer oder mehreren reinen virtuellen Funktionen werden als abstrakt betrachtet und können nicht instanziiert werden. Es können nur abgeleitete Klassen instanziiert werden, die Definitionen für alle rein virtuellen Funktionen definieren oder erben.

```
struct Abstract {
    virtual void f() = 0;
};

struct Concrete {
    void f() override {}
};

Abstract a; // Error.
Concrete c; // Good.
```

Selbst wenn eine Funktion als rein virtuell angegeben ist, kann eine Standardimplementierung angegeben werden. Trotzdem wird die Funktion als abstrakt betrachtet und abgeleitete Klassen müssen sie definieren, bevor sie instanziiert werden können. In diesem Fall darf die Version der abgeleiteten Klasse der Funktion sogar die Version der Basisklasse aufrufen.

```
struct DefaultAbstract {
    virtual void f() = 0;
};
void DefaultAbstract::f() {}

struct WhyWouldWeDoThis : DefaultAbstract {
    void f() override { DefaultAbstract::f(); }
};
```

Es gibt einige Gründe, warum wir dies tun möchten:

- Wenn wir eine Klasse erstellen möchten, die nicht selbst instanziiert werden kann, die instanziierten Klassen jedoch nicht davon abgehalten wird, können wir den Destruktor als rein virtuell deklarieren. Als Destruktor muss er auf jeden Fall definiert werden, wenn die Instanz freigegeben werden soll. Da der Destruktor höchstwahrscheinlich bereits virtuell ist, um Speicherverluste während der polymorphen Verwendung zu verhindern , wird es nicht unnötig sein, die Performance einer anderen `virtual` Funktion zu deklarieren. Dies kann bei der Erstellung von Schnittstellen hilfreich sein.

  ```
  struct Interface {
      virtual ~Interface() = 0;
  };
  Interface::~Interface() = default;

  struct Implementation : Interface {};
  // ~Implementation() is automatically defined by the compiler if not explicitly
  //  specified, meeting the "must be defined before instantiation" requirement.
  ```

- Wenn die meisten oder alle Implementierungen der reinen virtuellen Funktion doppelten Code enthalten, kann dieser Code stattdessen in die Basisklassenversion verschoben werden, wodurch die Pflege des Codes vereinfacht wird.

  ```
  class SharedBase {
      State my_state;
      std::unique_ptr<Helper> my_helper;
      // ...

    public:
      virtual void config(const Context& cont) = 0;
      // ...
  };
  /* virtual */ void SharedBase::config(const Context& cont) {
      my_helper = new Helper(my_state, cont.relevant_field);
      do_this();
      and_that();
  }

  class OneImplementation : public SharedBase {
      int i;
      // ...

    public:
      void config(const Context& cont) override;
      // ...
  };
  ```

```
void OneImplementation::config(const Context& cont) /* override */ {
    my_state = { cont.some_field, cont.another_field, i };
    SharedBase::config(cont);
    my_unique_setup();
};

// And so on, for other classes derived from SharedBase.
```

Virtuelle Elementfunktionen online lesen: https://riptutorial.com/de/cplusplus/topic/1752/virtuelle-elementfunktionen

Kapitel 141: Vorlagen

Einführung

Klassen, Funktionen und (seit C ++ 14) Variablen können erstellt werden. Eine Vorlage ist ein Stück Code mit einigen freien Parametern, die zu einer konkreten Klasse, Funktion oder Variablen werden, wenn alle Parameter angegeben werden. Parameter können Typen, Werte oder selbst Vorlagen sein. Eine bekannte Vorlage ist `std::vector` , die zu einem konkreten Containertyp wird, wenn der Elementtyp angegeben wird, *z. B.* `std::vector<int>` .

Syntax

- *Deklaration der* Vorlage < *template-parameter-list* >
- exportiere die *Deklaration* < *template-parameter-list* > / * bis C ++ 11 * /
- *Deklaration der* Vorlage <>
- Template - *Deklaration*
- extern Template - *Deklaration* / * da C ++ 11 * /
- Template < *Template-Parameter-List* > Klasse ... (*Opt*) *Bezeichner* (*Opt*)
- template *<template-Parameter-Liste> Klassenkennung (opt) = id-expression*
- template < *template-parameter-list* > typename ... (*opt*) *Bezeichner* (*opt*) / * seit C ++ 17 * /
- template *<template-Parameter-Liste>* Typname *Kennung (opt) = id-Ausdruck* / * da C ++ 17 * /
- *Postfix-Ausdruck* . Template- *ID-Ausdruck*
- *Postfix-Ausdruck* -> Vorlagen- *ID-Ausdruck*
- *Nested-Name-Specifier*- `template` *simple-template-id* `::`

Bemerkungen

Die `template` ist ein Schlüsselwort mit fünf verschiedenen Bedeutungen in der C ++ - Sprache, je nach Kontext.

1. Wenn eine Liste mit in `<>` eingeschlossenen Vorlagenparametern folgt, deklariert sie eine Vorlage, beispielsweise eine Klassenvorlage , eine Funktionsvorlage oder eine teilweise Spezialisierung einer vorhandenen Vorlage.

   ```
   template <class T>
   void increment(T& x) { ++x; }
   ```

2. Wenn von einem *leeren* `<>` gefolgt wird, wird eine explizite (vollständige) Spezialisierung angegeben .

   ```
   template <class T>
   void print(T x);

   template <> // <-- keyword used in this sense here
   ```

```
void print(const char* s) {
    // output the content of the string
    printf("%s\n", s);
}
```

3. Wenn eine Deklaration ohne `<>` folgt, bildet sie eine explizite Instantiierungsdeklaration oder -definition.

```
template <class T>
std::set<T> make_singleton(T x) { return std::set<T>(x); }

template std::set<int> make_singleton(int x); // <-- keyword used in this sense here
```

4. Innerhalb einer Vorlage Parameterliste, führt sie einen Template - Template - Parameter .

```
template <class T, template <class U> class Alloc>
//                 ^^^^^^^^ keyword used in this sense here
class List {
    struct Node {
        T value;
        Node* next;
    };
    Alloc<Node> allocator;
    Node* allocate_node() {
        return allocator.allocate(sizeof(T));
    }
    // ...
};
```

5. Nach dem Bereichsauflösungsoperator `::` und den Klassenmitgliedszugriffsoperatoren `.` und `->` gibt an, dass der folgende Name eine Vorlage ist.

```
struct Allocator {
    template <class T>
    T* allocate();
};

template <class T, class Alloc>
class List {
    struct Node {
        T value;
        Node* next;
    }
    Alloc allocator;
    Node* allocate_node() {
        // return allocator.allocate<Node>();       // error: < and > are interpreted as
                                                    // comparison operators
        return allocator.template allocate<Node>(); // ok; allocate is a template
        //               ^^^^^^^^ keyword used in this sense here
    }
};
```

Vor C ++ 11 konnte eine Vorlage mit dem `export` deklariert werden , um daraus eine *exportierte* Vorlage zu machen. Die Definition einer exportierten Vorlage muss nicht in jeder Übersetzungseinheit vorhanden sein, in der die Vorlage instanziiert wird. Zum Beispiel sollte

folgendes funktionieren:

`foo.h` :

```
#ifndef FOO_H
#define FOO_H
export template <class T> T identity(T x);
#endif
```

`foo.cpp` :

```
#include "foo.h"
template <class T> T identity(T x) { return x; }
```

`main.cpp` :

```
#include "foo.h"
int main() {
    const int x = identity(42); // x is 42
}
```

Aufgrund von Schwierigkeiten bei der Implementierung wurde das `export` von den meisten großen Compilern nicht unterstützt. Es wurde in C ++ 11 entfernt. Das `export` Schlüsselwort darf jetzt überhaupt nicht verwendet werden. Stattdessen müssen in der Regel Vorlagen in Kopfzeilen definiert werden (im Gegensatz zu Funktionen, die *keine* Vorlagen sind und normalerweise *nicht* in Kopfzeilen definiert sind). Siehe Warum können Vorlagen nur in der Headerdatei implementiert werden?

Examples

Funktionsvorlagen

Das Templating kann auch auf Funktionen (wie auch auf traditionelle Strukturen) mit dem gleichen Effekt angewendet werden.

```
// 'T' stands for the unknown type
// Both of our arguments will be of the same type.
template<typename T>
void printSum(T add1, T add2)
{
    std::cout << (add1 + add2) << std::endl;
}
```

Diese kann dann wie Strukturvorlagen verwendet werden.

```
printSum<int>(4, 5);
printSum<float>(4.5f, 8.9f);
```

In beiden Fällen wird das Template-Argument verwendet, um die Typen der Parameter zu ersetzen. Das Ergebnis funktioniert wie eine normale C ++ - Funktion (wenn die Parameter nicht

mit dem Vorlagentyp übereinstimmen, wendet der Compiler die Standardkonvertierungen an).

Eine weitere Eigenschaft von Template-Funktionen (im Gegensatz zu Template-Klassen) besteht darin, dass der Compiler die Template-Parameter anhand der an die Funktion übergebenen Parameter ableiten kann.

```
printSum(4, 5);     // Both parameters are int.
                    // This allows the compiler deduce that the type
                    // T is also int.

printSum(5.0, 4);   // In this case the parameters are two different types.
                    // The compiler is unable to deduce the type of T
                    // because there are contradictions. As a result
                    // this is a compile time error.
```

Mit dieser Funktion können wir den Code vereinfachen, wenn wir Vorlagenstrukturen und Funktionen kombinieren. In der Standardbibliothek gibt es ein gemeinsames Muster, mit dem wir die `template structure X` mit der `make_X()` .

```
// The make_X pattern looks like this.
// 1) A template structure with 1 or more template types.
template<typename T1, typename T2>
struct MyPair
{
    T1      first;
    T2      second;
};
// 2) A make function that has a parameter type for
//    each template parameter in the template structure.
template<typename T1, typename T2>
MyPair<T1, T2> make_MyPair(T1 t1, T2 t2)
{
    return MyPair<T1, T2>{t1, t2};
}
```

Wie hilft das?

```
auto val1 = MyPair<int, float>{5, 8.7};     // Create object explicitly defining the types
auto val2 = make_MyPair(5, 8.7);            // Create object using the types of the paramters.
                                            // In this code both val1 and val2 are the same
                                            // type.
```

Hinweis: Dies ist nicht dazu gedacht, den Code zu verkürzen. Dies soll den Code robuster machen. Dadurch können die Typen geändert werden, indem der Code an einer einzelnen Stelle und nicht an mehreren Stellen geändert wird.

Weiterleitung von Argumenten

Vorlage akzeptiert möglicherweise lvalue- und rvalue-Referenzen unter Verwendung der *Weiterleitungsreferenz* :

```
template <typename T>
void f(T &&t);
```

In diesem Fall wird der tatsächliche Typ von `t` abhängig vom Kontext hergeleitet:

```
struct X { };

X x;
f(x); // calls f<X&>(x)
f(X()); // calls f<X>(x)
```

Im ersten Fall wird der Typ `T` als *Verweis auf* `X` (`X&`) hergeleitet, und der Typ von `t` ist der *Wertewert auf* `X` , während im zweiten Fall der Typ von `T` als `X` und der Typ von `t` als *Wertewert abgeleitet wird zu* `X` (`X&&`).

Hinweis: Beachten Sie, dass `decltype(t)` im ersten Fall dasselbe wie `T` ist, im zweiten Fall jedoch nicht.

Um `t` perfekt an eine andere Funktion weiterzuleiten, unabhängig davon, ob es sich um eine lvalue- oder eine rvalue-Referenz handelt, muss `std::forward` :

```
template <typename T>
void f(T &&t) {
    g(std::forward<T>(t));
}
```

Weiterleitungsreferenzen können mit variadischen Vorlagen verwendet werden:

```
template <typename... Args>
void f(Args&&... args) {
    g(std::forward<Args>(args)...);
}
```

Hinweis: Weiterleitungsreferenzen können nur für Vorlagenparameter verwendet werden, z. B. ist im folgenden Code `v` eine rvalue-Referenz und keine Weiterleitungsreferenz:

```
#include <vector>

template <typename T>
void f(std::vector<T> &&v);
```

Grundlegende Klassenvorlage

Die Grundidee einer Klassenvorlage ist, dass der Vorlagenparameter zur Kompilierzeit durch einen Typ ersetzt wird. Das Ergebnis ist, dass dieselbe Klasse für mehrere Typen wiederverwendet werden kann. Der Benutzer gibt an, welcher Typ verwendet wird, wenn eine Variable der Klasse deklariert wird. Drei Beispiele dafür sind in `main()` :

```
#include <iostream>
using std::cout;

template <typename T>         // A simple class to hold one number of any type
class Number {
public:
```

```
    void setNum(T n);         // Sets the class field to the given number
    T plus1() const;          // returns class field's "follower"
private:
    T num;                    // Class field
};

template <typename T>         // Set the class field to the given number
void Number<T>::setNum(T n) {
    num = n;
}

template <typename T>         // returns class field's "follower"
T Number<T>::plus1() const {
    return num + 1;
}

int main() {
    Number<int> anInt;        // Test with an integer (int replaces T in the class)
    anInt.setNum(1);
    cout << "My integer + 1 is " << anInt.plus1() << "\n";     // Prints 2

    Number<double> aDouble;   // Test with a double
    aDouble.setNum(3.1415926535897);
    cout << "My double + 1 is " << aDouble.plus1() << "\n";    // Prints 4.14159

    Number<float> aFloat;     // Test with a float
    aFloat.setNum(1.4);
    cout << "My float + 1 is " << aFloat.plus1() << "\n";      // Prints 2.4

    return 0;  // Successful completion
}
```

Template-Spezialisierung

Sie können die Implementierung für bestimmte Instanziierungen einer Vorlagenklasse / -methode definieren.

Zum Beispiel, wenn Sie haben:

```
template <typename T>
T sqrt(T t) { /* Some generic implementation */ }
```

Sie können dann schreiben:

```
template<>
int sqrt<int>(int i) { /* Highly optimized integer implementation */ }
```

Ein Benutzer, der `sqrt(4.0)` schreibt, erhält dann die generische Implementierung, während `sqrt(4)` die spezialisierte Implementierung erhält.

Teilweise Schablonenspezialisierung

Im Gegensatz zu einer vollständigen Schablonenspezialisierung ermöglicht die partielle Schablonenspezialisierung die Einführung einer Schablone mit einigen der Argumente der

vorhandenen Schablone. Die teilweise Vorlagenspezialisierung ist nur für Vorlagenklassen / -strukturen verfügbar:

```
// Common case:
template<typename T, typename U>
struct S {
    T t_val;
    U u_val;
};

// Special case when the first template argument is fixed to int
template<typename V>
struct S<int, V> {
    double another_value;
    int foo(double arg) {// Do something}
};
```

Wie oben gezeigt, können partielle Vorlagenspezialisierungen völlig unterschiedliche Datensätze und Funktionsmitglieder einführen.

Wenn eine teilweise spezialisierte Vorlage instanziiert wird, wird die am besten geeignete Spezialisierung ausgewählt. Zum Beispiel definieren wir eine Vorlage und zwei Teilspezialisierungen:

```
template<typename T, typename U, typename V>
struct S {
    static void foo() {
        std::cout << "General case\n";
    }
};

template<typename U, typename V>
struct S<int, U, V> {
    static void foo() {
        std::cout << "T = int\n";
    }
};

template<typename V>
struct S<int, double, V> {
    static void foo() {
        std::cout << "T = int, U = double\n";
    }
};
```

Nun die folgenden Anrufe:

```
S<std::string, int, double>::foo();
S<int, float, std::string>::foo();
S<int, double, std::string>::foo();
```

wird drucken

```
General case
T = int
```

```
T = int, U = double
```

Funktionsvorlagen dürfen nur vollständig spezialisiert sein:

```
template<typename T, typename U>
void foo(T t, U u) {
    std::cout << "General case: " << t << " " << u << std::endl;
}

// OK.
template<>
void foo<int, int>(int a1, int a2) {
    std::cout << "Two ints: " << a1 << " " << a2 << std::endl;
}

void invoke_foo() {
    foo(1, 2.1); // Prints "General case: 1 2.1"
    foo(1,2);    // Prints "Two ints: 1 2"
}

// Compilation error: partial function specialization is not allowed.
template<typename U>
void foo<std::string, U>(std::string t, U u) {
    std::cout << "General case: " << t << " " << u << std::endl;
}
```

Standard-Vorlagenparameterwert

Wie bei den Funktionsargumenten können Vorlagenparameter ihre Standardwerte haben. Alle Vorlagenparameter mit einem Standardwert müssen am Ende der Vorlagenparameterliste deklariert werden. Die Grundidee ist, dass die Vorlagenparameter mit dem Standardwert während der Instantiierung der Vorlage weggelassen werden können.

Einfaches Beispiel für die Verwendung von Standardvorlagenparameterwerten:

```
template <class T, size_t N = 10>
struct my_array {
    T arr[N];
};

int main() {
    /* Default parameter is ignored, N = 5 */
    my_array<int, 5> a;

    /* Print the length of a.arr: 5 */
    std::cout << sizeof(a.arr) / sizeof(int) << std::endl;

    /* Last parameter is omitted, N = 10 */
    my_array<int> b;

    /* Print the length of a.arr: 10 */
    std::cout << sizeof(b.arr) / sizeof(int) << std::endl;
}
```

Alias-Vorlage

C ++ 11

Grundbeispiel:

```
template<typename T> using pointer = T*;
```

Diese Definition macht den `pointer<T>` einem Alias von `T*` . Zum Beispiel:

```
pointer<int> p = new int; // equivalent to: int* p = new int;
```

Aliasvorlagen können nicht spezialisiert werden. Diese Funktionalität kann jedoch indirekt erhalten werden, indem sie sich auf einen verschachtelten Typ in einer Struktur beziehen:

```
template<typename T>
 struct nonconst_pointer_helper { typedef T* type; };

template<typename T>
 struct nonconst_pointer_helper<T const> { typedef T* type; };

template<typename T> using nonconst_pointer = nonconst_pointer_helper<T>::type;
```

Parameter für Vorlagenvorlagen

Manchmal möchten wir einen Schablonentyp in die Vorlage einreichen, ohne dessen Werte festzulegen. Dafür werden Vorlagenvorlagenparameter erstellt. Sehr einfache Vorlagenparameterbeispiele:

```
template <class T>
struct Tag1 { };

template <class T>
struct Tag2 { };

template <template <class> class Tag>
struct IntTag {
   typedef Tag<int> type;
};

int main() {
   IntTag<Tag1>::type t;
}
```

C ++ 11

```
#include <vector>
#include <iostream>

template <class T, template <class...> class C, class U>
C<T> cast_all(const C<U> &c) {
   C<T> result(c.begin(), c.end());
   return result;
}

int main() {
   std::vector<float> vf = {1.2, 2.6, 3.7};
```

```
   auto vi = cast_all<int>(vf);
   for(auto &&i: vi) {
      std::cout << i << std::endl;
   }
}
```

Nicht-Typ-Vorlagenargumente mit auto deklarieren

Vor C ++ 17 mussten Sie beim Schreiben eines nicht typisierten Parameters der Vorlage zuerst seinen Typ angeben. So wurde ein allgemeines Muster zum Schreiben:

```
template <class T, T N>
struct integral_constant {
    using type = T;
    static constexpr T value = N;
};

using five = integral_constant<int, 5>;
```

Für komplizierte Ausdrücke bedeutet dies jedoch, dass Sie `decltype(expr), expr` schreiben `decltype(expr), expr` wenn Sie Templates instantiieren. Die Lösung besteht darin, diese Sprache zu vereinfachen und einfach `auto` zuzulassen:

C ++ 17

```
template <auto N>
struct integral_constant {
    using type = decltype(N);
    static constexpr type value = N;
};

using five = integral_constant<5>;
```

Leeres benutzerdefiniertes Deleter für unique_ptr

Ein schönes motivierendes Beispiel kann der Versuch sein, die leere Basisoptimierung mit einem benutzerdefinierten Deleter für `unique_ptr` zu kombinieren. Verschiedene C-API-Deleter haben unterschiedliche Rückgabetypen, aber das ist uns egal - wir möchten nur, dass für jede Funktion etwas funktioniert:

```
template <auto DeleteFn>
struct FunctionDeleter {
    template <class T>
    void operator()(T* ptr) const {
        DeleteFn(ptr);
    }
};

template <T, auto DeleteFn>
using unique_ptr_deleter = std::unique_ptr<T, FunctionDeleter<DeleteFn>>;
```

Und jetzt können Sie einfach einen beliebigen Funktionszeiger verwenden, der ein Argument des

Typs `T` unabhängig vom Rückgabetyp als nicht-typisierten Parameter für die Vorlage übernehmen kann und daraus einen `unique_ptr` die Größe `unique_ptr` erhält:

```
unique_ptr_deleter<std::FILE, std::fclose> p;
```

Nicht-Typ-Vorlagenparameter

Abgesehen von Types als Template-Parameter dürfen wir Werte von konstanten Ausdrücken deklarieren, die eines der folgenden Kriterien erfüllen:

- Integral- oder Aufzählungstyp,
- Zeiger auf Objekt oder Zeiger auf Funktion
- lWertreferenz auf Objekt oder lWertreferenz auf Funktion,
- Zeiger auf Mitglied,
- `std::nullptr_t` .

Wie alle Vorlagenparameter können Vorlagenparameter, die nicht vom Typ sind, explizit angegeben, vorgegeben oder implizit über die Vorlagen-Argumentableitung abgeleitet werden.

Beispiel für die Verwendung von nicht typisierten Vorlagenparametern:

```
#include <iostream>

template<typename T, std::size_t size>
std::size_t size_of(T (&anArray)[size])  // Pass array by reference. Requires.
{                                        // an exact size. We allow all sizes
    return size;                         // by using a template "size".
}

int main()
{
    char anArrayOfChar[15];
    std::cout << "anArrayOfChar: " << size_of(anArrayOfChar) << "\n";

    int  anArrayOfData[] = {1,2,3,4,5,6,7,8,9};
    std::cout << "anArrayOfData: " << size_of(anArrayOfData) << "\n";
}
```

Beispiel für die explizite Angabe von Vorlagenparametern vom Typ und nicht vom Typ:

```
#include <array>
int main ()
{
    std::array<int, 5> foo; // int is a type parameter, 5 is non-type
}
```

Nicht-Typ-Vorlagenparameter sind eine der Möglichkeiten, um die Wiederholung von Vorlagen zu erreichen, und ermöglichen die Durchführung von Metaprogrammierungen .

Variadische Vorlagendatenstrukturen

C ++ 14

Es ist häufig nützlich, Klassen oder Strukturen zu definieren, die eine variable Anzahl und einen Typ von Datenelementen haben, die zur Kompilierzeit definiert werden. Das kanonische Beispiel ist `std::tuple` . In manchen `std::tuple` müssen Sie jedoch Ihre eigenen benutzerdefinierten Strukturen definieren. Hier ist ein Beispiel, in dem die Struktur mithilfe von Compounding definiert wird (anstelle von Vererbung wie bei `std::tuple` . Beginnen Sie mit der allgemeinen (leeren) Definition, die in der späteren Spezialisierung auch als Basisfall für die Beendigung der Rekrutierung dient:

```
template<typename ... T>
struct DataStructure {};
```

Dies erlaubt uns bereits, eine leere Struktur, `DataStructure<> data` , zu definieren, obwohl dies noch nicht sehr nützlich ist.

Als nächstes kommt die rekursive Fallspezialisierung:

```
template<typename T, typename ... Rest>
struct DataStructure<T, Rest ...>
{
    DataStructure(const T& first, const Rest& ... rest)
        : first(first)
        , rest(rest...)
    {}

    T first;
    DataStructure<Rest ... > rest;
};
```

Dies reicht nun aus, um beliebige Datenstrukturen wie `DataStructure<int, float, std::string> data(1, 2.1, "hello")` zu erstellen.

So was ist los? Beachten Sie zunächst, dass es sich hierbei um eine Spezialisierung handelt, deren Anforderung es ist, dass mindestens ein variadischer Vorlagenparameter (nämlich `T` oben) vorhanden ist, ohne sich um die spezifische Zusammensetzung der Packung `Rest` kümmern. Das Wissen , dass `T` existiert erlaubt die Definition seines Datenelements `first` . Der Rest der Daten wird rekursiv als `DataStructure<Rest ... > rest` . Der Konstruktor initiiert beide Member, einschließlich eines rekursiven Konstruktoraufrufs für das `rest` Member.

Um dies besser zu verstehen, können wir ein Beispiel `DataStructure<int, float> data` : Angenommen, Sie haben eine Deklaration `DataStructure<int, float> data` . Die Deklaration stimmt zuerst mit der Spezialisierung überein und ergibt eine Struktur mit den `DataStructure<float> rest` `int first` und `DataStructure<float> rest` . Der `rest` Definition paßt wieder diese Spezialisierung, seinen eigenen Schaffung `float first` und `DataStructure<> rest` Mitglieder. Diese letzte `rest` entspricht schließlich der Definition des Basisfalls und erzeugt eine leere Struktur.

Sie können dies wie folgt visualisieren:

```
DataStructure<int, float>
   -> int first
   -> DataStructure<float> rest
         -> float first
```

```
         -> DataStructure<> rest
              -> (empty)
```

Nun haben wir die Datenstruktur, die jedoch noch nicht besonders nützlich ist, da wir nicht einfach auf die einzelnen Datenelemente zugreifen können (um beispielsweise auf das letzte Mitglied der `DataStructure<int, float, std::string> data` wir `data.rest.rest.first` , was nicht gerade benutzerfreundlich ist). So fügen wir eine `get` - Methode , um es (nur in der Spezialisierung erforderlich , da die Basis-Gehäusestruktur keine Daten zu `get`):

```
template<typename T, typename ... Rest>
struct DataStructure<T, Rest ...>
{
    ...
    template<size_t idx>
    auto get()
    {
        return GetHelper<idx, DataStructure<T,Rest...>>::get(*this);
    }
    ...
};
```

Wie Sie sehen, wird diese `get` Member-Funktion selbst erstellt - diesmal auf dem Index des Member, das benötigt wird (die Verwendung kann z. B. `data.get<1>()` , ähnlich wie bei `std::tuple`). Die eigentliche Arbeit wird von einer statischen Funktion in der `GetHelper` . Der Grund, warum wir die erforderliche Funktionalität nicht direkt in `DataStructure` Definition definieren können `get` liegt darin, dass wir (wie wir bald sehen werden) uns auf `idx` spezialisieren `idx` Es ist jedoch nicht möglich, eine Template-Member-Funktion zu spezialisieren, ohne die enthaltende Klasse zu spezialisieren Vorlage. Beachten Sie auch, dass die Verwendung eines `auto` C ++ 14-Stil hier unser Leben erheblich vereinfacht, da wir sonst einen komplizierten Ausdruck für den Rückgabetyp benötigen.

Also weiter zur Helferklasse. Diesmal benötigen wir eine leere Vorwärtsdeklaration und zwei Spezialisierungen. Zuerst die Erklärung:

```
template<size_t idx, typename T>
struct GetHelper;
```

Nun der Basisfall (wenn `idx==0`). In diesem Fall schicken wir einfach das `first` Mitglied zurück:

```
template<typename T, typename ... Rest>
struct GetHelper<0, DataStructure<T, Rest ... >>
{
    static T get(DataStructure<T, Rest...>& data)
    {
        return data.first;
    }
};
```

In dem rekursiven Fall verringern wir `idx` und das Aufrufen `GetHelper` für den `rest` Mitglied:

```
template<size_t idx, typename T, typename ... Rest>
```

```
struct GetHelper<idx, DataStructure<T, Rest ... >>
{
    static auto get(DataStructure<T, Rest...>& data)
    {
        return GetHelper<idx-1, DataStructure<Rest ...>>::get(data.rest);
    }
};
```

Angenommen, wir haben `DataStructure<int, float> data` `data.get<1>()` und benötigen `data.get<1>()` . Das ruft `GetHelper<1, DataStructure<int, float>>::get(data)` (die zweite Spezialisierung) auf, das wiederum `GetHelper<0, DataStructure<float>>::get(data.rest)` , das schließlich zurückkehrt (von der 1. Spezialisierung, da jetzt `idx` 0 ist) `data.rest.first` .

So, das war es! Hier ist der ganze Funktion Code, mit einigem Beispiel für die Verwendung in der `main`

```
#include <iostream>

template<size_t idx, typename T>
struct GetHelper;

template<typename ... T>
struct DataStructure
{
};

template<typename T, typename ... Rest>
struct DataStructure<T, Rest ...>
{
    DataStructure(const T& first, const Rest& ... rest)
        : first(first)
        , rest(rest...)
    {}

    T first;
    DataStructure<Rest ... > rest;

    template<size_t idx>
    auto get()
    {
        return GetHelper<idx, DataStructure<T,Rest...>>::get(*this);
    }
};

template<typename T, typename ... Rest>
struct GetHelper<0, DataStructure<T, Rest ... >>
{
    static T get(DataStructure<T, Rest...>& data)
    {
        return data.first;
    }
};

template<size_t idx, typename T, typename ... Rest>
struct GetHelper<idx, DataStructure<T, Rest ... >>
{
    static auto get(DataStructure<T, Rest...>& data)
    {
        return GetHelper<idx-1, DataStructure<Rest ...>>::get(data.rest);
```

```
    }
};

int main()
{
    DataStructure<int, float, std::string> data(1, 2.1, "Hello");

    std::cout << data.get<0>() << std::endl;
    std::cout << data.get<1>() << std::endl;
    std::cout << data.get<2>() << std::endl;

    return 0;
}
```

Explizite Instantiierung

Eine explizite Instantiierungsdefinition erstellt und deklariert eine konkrete Klasse, Funktion oder Variable aus einer Vorlage, ohne sie noch zu verwenden. Auf eine explizite Instanziierung kann von anderen Übersetzungseinheiten aus verwiesen werden. Dies kann verwendet werden, um die Definition einer Vorlage in einer Header-Datei zu vermeiden, wenn diese nur mit einer begrenzten Anzahl von Argumenten instanziiert wird. Zum Beispiel:

```
// print_string.h
template <class T>
void print_string(const T* str);

// print_string.cpp
#include "print_string.h"
template void print_string(const char*);
template void print_string(const wchar_t*);
```

Da `print_string<char>` und `print_string<wchar_t>` in `print_string.cpp` explizit instanziiert `print_string.cpp` , kann der Linker sie auch finden, obwohl die Vorlage `print_string` nicht im Header definiert ist. Wenn diese expliziten Instanziierungsdeklarationen nicht vorhanden wären, würde wahrscheinlich ein Linker-Fehler auftreten. Siehe Warum können Vorlagen nur in der Headerdatei implementiert werden?

C ++ 11

Wenn eine explizite Instanziierung Definition durch das vorangestellt ist `extern` Schlüsselwort , wird es eine explizite Instanziierung *Deklaration* statt. Das Vorhandensein einer expliziten Instantiierungsdeklaration für eine bestimmte Spezialisierung verhindert die implizite Instantiierung der jeweiligen Spezialisierung innerhalb der aktuellen Übersetzungseinheit. Ein Verweis auf diese Spezialisierung, der andernfalls eine implizite Instantiierung verursachen würde, kann stattdessen auf eine explizite Instantiierungsdefinition in derselben oder einer anderen TU verweisen.

`foo.h`

```
#ifndef FOO_H
#define FOO_H
template <class T> void foo(T x) {
    // complicated implementation
}
```

```
#endif
```

foo.cpp

```
#include "foo.h"
// explicit instantiation definitions for common cases
template void foo(int);
template void foo(double);
```

main.cpp

```
#include "foo.h"
// we already know foo.cpp has explicit instantiation definitions for these
extern template void foo(double);
int main() {
    foo(42);   // instantiates foo<int> here;
               // wasteful since foo.cpp provides an explicit instantiation already!
    foo(3.14); // does not instantiate foo<double> here;
               // uses instantiation of foo<double> in foo.cpp instead
}
```

Vorlagen online lesen: https://riptutorial.com/de/cplusplus/topic/460/vorlagen

Kapitel 142: Weitere undefinierte Verhalten in C ++

Einführung

Weitere Beispiele, wie C ++ schief gehen kann.

Fortsetzung von undefiniertem Verhalten

Examples

Verweise auf nicht statische Member in Initialisierungslisten

Verweise auf nicht statische Member in Initialisierungslisten, bevor der Konstruktor mit der Ausführung begonnen hat, kann zu undefiniertem Verhalten führen. Dies ergibt sich, da nicht alle Mitglieder zu diesem Zeitpunkt aufgebaut sind. Aus dem Standardentwurf:

> §12.7.1: Bei einem Objekt mit einem nicht trivialen Konstruktor führt der Verweis auf ein nicht statisches Member oder eine beliebige Basisklasse des Objekts, bevor der Konstruktor mit der Ausführung beginnt, zu undefiniertem Verhalten.

Beispiel

```
struct W { int j; };
struct X : public virtual W { };
struct Y {
   int *p;
   X x;
   Y() : p(&x.j) { // undefined, x is not yet constructed
   }
};
```

Weitere undefinierte Verhalten in C ++ online lesen:
https://riptutorial.com/de/cplusplus/topic/9885/weitere-undefinierte-verhalten-in-c-plusplus

Kapitel 143: Wert und Referenzsemantik

Examples

Tiefe Kopier- und Bewegungsunterstützung

Wenn ein Typ Wertsemantik haben möchte und Objekte, die dynamisch zugewiesen werden, speichern muss, muss der Typ bei Kopiervorgängen neue Kopien dieser Objekte zuordnen. Dies muss auch für die Kopierzuweisung erfolgen.

Diese Art des Kopierens wird als "tiefe Kopie" bezeichnet. Es nimmt effektiv, was sonst eine Referenzsemantik gewesen wäre, und wandelt sie in eine Wertesemantik um:

```
struct Inner {int i;};

const int NUM_INNER = 5;
class Value
{
private:
  Inner *array_; //Normally has reference semantics.

public:
  Value() : array_(new Inner[NUM_INNER]){}

  ~Value() {delete[] array_;}

  Value(const Value &val) : array_(new Inner[NUM_INNER])
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
  }

  Value &operator=(const Value &val)
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
    return *this;
  }
};
```

C ++ 11

Verschiebungs-Semantik erlaubt einem Typ wie `Value` , um zu vermeiden, dass die referenzierten Daten wirklich kopiert werden. Wenn der Benutzer den Wert auf eine Weise verwendet, die eine Verschiebung auslöst, kann das "kopierte" Objekt von den Daten, auf die es verweist, leer bleiben:

```
struct Inner {int i;};

constexpr auto NUM_INNER = 5;
class Value
{
private:
  Inner *array_; //Normally has reference semantics.
```

```
public:
  Value() : array_(new Inner[NUM_INNER]){}

  //OK to delete even if nullptr
  ~Value() {delete[] array_;}

  Value(const Value &val) : array_(new Inner[NUM_INNER])
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
  }

  Value &operator=(const Value &val)
  {
    for(int i = 0; i < NUM_INNER; ++i)
      array_[i] = val.array_[i];
    return *this;
  }

  //Movement means no memory allocation.
  //Cannot throw exceptions.
  Value(Value &&val) noexcept : array_(val.array_)
  {
    //We've stolen the old value.
    val.array_ = nullptr;
  }

  //Cannot throw exceptions.
  Value &operator=(Value &&val) noexcept
  {
    //Clever trick. Since `val` is going to be destroyed soon anyway,
    //we swap his data with ours. His destructor will destroy our data.
    std::swap(array_, val.array_);
  }
};
```

In der Tat können wir einen solchen Typ sogar nicht kopierfähig machen, wenn wir tiefe Kopien verbieten möchten, während das Objekt dennoch verschoben werden kann.

```
struct Inner {int i;};

constexpr auto NUM_INNER = 5;
class Value
{
private:
  Inner *array_; //Normally has reference semantics.

public:
  Value() : array_(new Inner[NUM_INNER]){}

  //OK to delete even if nullptr
  ~Value() {delete[] array_;}

  Value(const Value &val) = delete;
  Value &operator=(const Value &val) = delete;

  //Movement means no memory allocation.
  //Cannot throw exceptions.
  Value(Value &&val) noexcept : array_(val.array_)
```

```
  {
    //We've stolen the old value.
    val.array_ = nullptr;
  }

  //Cannot throw exceptions.
  Value &operator=(Value &&val) noexcept
  {
    //Clever trick. Since `val` is going to be destroyed soon anyway,
    //we swap his data with ours. His destructor will destroy our data.
    std::swap(array_, val.array_);
  }
};
```

Wir können sogar die Nullregel anwenden, indem Sie `unique_ptr` :

```
struct Inner {int i;};

constexpr auto NUM_INNER = 5;
class Value
{
private:
  unique_ptr<Inner []>array_; //Move-only type.

public:
  Value() : array_(new Inner[NUM_INNER]){}

  //No need to explicitly delete. Or even declare.
  ~Value() = default; {delete[] array_;}

  //No need to explicitly delete. Or even declare.
  Value(const Value &val) = default;
  Value &operator=(const Value &val) = default;

  //Will perform an element-wise move.
  Value(Value &&val) noexcept = default;

  //Will perform an element-wise move.
  Value &operator=(Value &&val) noexcept = default;
};
```

Definitionen

Ein Typ hat eine Wertesemantik, wenn sich der beobachtbare Zustand des Objekts von allen anderen Objekten dieses Typs funktional unterscheidet. Wenn Sie also ein Objekt kopieren, haben Sie ein neues Objekt, und Änderungen am neuen Objekt sind vom alten Objekt aus nicht sichtbar.

Die meisten grundlegenden C ++ - Typen haben eine Wertesemantik:

```
int i = 5;
int j = i; //Copied
j += 20;
std::cout << i; //Prints 5; i is unaffected by changes to j.
```

Die meisten in der Standardbibliothek definierten Typen haben auch eine Wertesemantik:

```
std::vector<int> v1(5, 12); //array of 5 values, 12 in each.
std::vector<int> v2 = v1; //Copies the vector.
v2[3] = 6; v2[4] = 9;
std::cout << v1[3] << " " << v1[4]; //Writes "12 12", since v1 is unchanged.
```

Von einem Typ wird gesagt, dass er eine Referenzsemantik hat, wenn eine Instanz dieses Typs seinen beobachtbaren Status mit einem anderen Objekt (außerhalb des Objekts) gemeinsam nutzen kann, sodass durch das Bearbeiten eines Objekts der Status innerhalb eines anderen Objekts geändert wird.

C ++ - Zeiger haben eine Wertesemantik in Bezug auf das Objekt, auf das sie zeigen, aber sie haben Referenzsemantik in Bezug auf den *Zustand* des Objekts, auf das sie zeigen:

```
int *pi = new int(4);
int *pi2 = pi;
pi = new int(16);
assert(pi2 != pi); //Will always pass.

int *pj = pi;
*pj += 5;
std::cout << *pi; //Writes 9, since `pi` and `pj` reference the same object.
```

C ++ - Referenzen haben auch Referenzsemantiken.

Wert und Referenzsemantik online lesen: https://riptutorial.com/de/cplusplus/topic/1955/wert-und-referenzsemantik

Kapitel 144: Wertkategorien

Examples

Bedeutungen der Wertkategorie

Ausdrücken in C ++ wird basierend auf dem Ergebnis dieser Ausdrücke eine bestimmte Wertkategorie zugewiesen. Wertkategorien für Ausdrücke können die Auflösung der C ++ -Funktion beeinflussen.

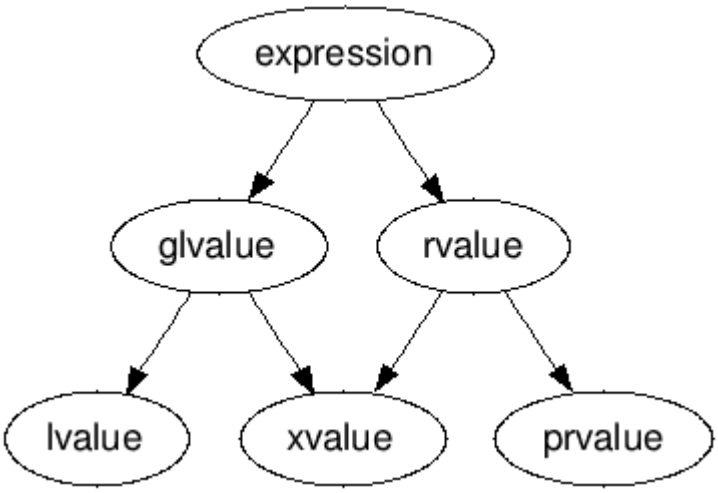
Wertkategorien bestimmen zwei wichtige, aber getrennte Eigenschaften eines Ausdrucks. Eine Eigenschaft ist, ob der Ausdruck eine Identität hat. Ein Ausdruck hat Identität, wenn er sich auf ein Objekt bezieht, das einen Variablennamen hat. Der Variablenname ist möglicherweise nicht in den Ausdruck einbezogen, das Objekt kann jedoch einen enthalten.

Die andere Eigenschaft ist, ob es legal ist, implizit vom Wert des Ausdrucks abzuweichen. Oder genauer gesagt, ob der Ausdruck, wenn er als Funktionsparameter verwendet wird, an R-Wert-Parametertypen bindet oder nicht.

C ++ definiert drei Wertkategorien, die die nützliche Kombination dieser Eigenschaften darstellen: lWert (Ausdrücke mit Identität, aber nicht aus, die verschoben werden können), xWert (Ausdrücke mit Identität, aus denen sich beweglich kann) und prvalue (Ausdrücke ohne Identität, aus denen sich verschieben lässt). C ++ hat keine Ausdrücke, die keine Identität haben und nicht verschoben werden können.

C ++ definiert zwei andere Wertkategorien, die jeweils ausschließlich auf einer dieser Eigenschaften basieren: glvalue (Ausdrücke mit Identität) und rvalue (Ausdrücke, aus denen verschoben werden kann). Diese dienen als nützliche Gruppierungen der vorherigen Kategorien.

Diese Grafik dient als Illustration:



prvalue

Ein prvalue-Ausdruck (pure-rvalue) ist ein Ausdruck, dem die Identität fehlt, dessen Auswertung normalerweise zur Initialisierung eines Objekts verwendet wird und aus dem implizit verschoben werden kann. Dazu gehören unter anderem:

- Ausdrücke, die temporäre Objekte darstellen, z. B. `std::string("123")` .
- Ein Funktionsaufrufsausdruck, der keinen Verweis zurückgibt
- Ein Literal (*außer* einem String-Literal - das sind lvalues), wie `1` , `true` , `0.5f` oder `'a'`
- Ein Lambda-Ausdruck

Der integrierte addressof-Operator (`&`) kann nicht auf diese Ausdrücke angewendet werden.

xvalue

Ein xvalue (eXpiring value) -Ausdruck ist ein Ausdruck, der eine Identität hat und ein Objekt darstellt, aus dem implizit verschoben werden kann. Die allgemeine Idee mit xvalue-Ausdrücken ist, dass das Objekt, das sie repräsentieren, bald zerstört werden wird (daher der "eXpiring" -Teil) und daher implizit ein Abgehen von ihnen in Ordnung ist.

Gegeben:

```
struct X { int n; };
extern X x;

4;                    // prvalue: does not have an identity
x;                    // lvalue
x.n;                  // lvalue
std::move(x);         // xvalue
std::forward<X&>(x); // lvalue
X{4};                 // prvalue: does not have an identity
X{4}.n;               // xvalue: does have an identity and denotes resources
                      // that can be reused
```

lWert

Ein lvalue-Ausdruck ist ein Ausdruck, der eine Identität hat, jedoch nicht implizit verschoben werden kann. Dazu gehören Ausdrücke, die aus einem Variablennamen, Funktionsnamen, Ausdrücken, die integrierte Dereferenzierungsoperatoren sind, und Ausdrücken bestehen, die auf lvalue-Referenzen verweisen.

Der typische Wert ist einfach ein Name, aber Werte können auch in anderen Geschmacksrichtungen verwendet werden:

```
struct X { ... };

X x;         // x is an lvalue
X* px = &x;  // px is an lvalue
*px = X{};   // *px is also an lvalue, X{} is a prvalue

X* foo_ptr();  // foo_ptr() is a prvalue
X& foo_ref();  // foo_ref() is an lvalue
```

Während die meisten Literale (z. B. `4` , `'x'` usw.) Werte sind, sind String-Literale Werte.

glvalue

Ein glvalue-Ausdruck (ein "generalisierter lvalue") - Ausdruck ist jeder Ausdruck, der eine Identität hat, unabhängig davon, ob er verschoben werden kann oder nicht. Diese Kategorie umfasst l-Werte (Ausdrücke, die eine Identität haben, aus denen jedoch kein Platz verschoben werden kann) und x-Werte (Ausdrücke, die eine Identität haben und aus denen verschoben werden kann), schließt jedoch keine pr-Werte (Ausdrücke ohne Identität) aus.

Wenn ein Ausdruck einen *Namen hat* , ist dies ein guter Wert:

```
struct X { int n; };
X foo();

X x;
x; // has a name, so it's a glvalue
std::move(x); // has a name (we're moving from "x"), so it's a glvalue
              // can be moved from, so it's an xvalue not an lvalue

foo(); // has no name, so is a prvalue, not a glvalue
X{};   // temporary has no name, so is a prvalue, not a glvalue
X{}.n; // HAS a name, so is a glvalue. can be moved from, so it's an xvalue
```

Wert

Ein rvalue-Ausdruck ist ein Ausdruck, aus dem implizit verschoben werden kann, unabhängig davon, ob er eine Identität hat.

Genauer gesagt, können rvalue-Ausdrücke als Argument für eine Funktion verwendet werden, die einen Parameter des Typs `T &&` (wobei `T` der Typ des `expr`) verwendet. *Nur* Argumentausdrücke können als Argument für solche Funktionsparameter angegeben werden. Wenn ein nicht rvalue-Ausdruck verwendet wird, wählt die Überladungsauflösung eine Funktion aus, die keinen rvalue-Referenzparameter verwendet. Und wenn keine vorhanden sind, wird ein Fehler angezeigt.

Die Kategorie von rvalue-Ausdrücken umfasst alle xvalue- und prvalue-Ausdrücke und nur diese Ausdrücke.

Die Standardbibliotheksfunktion `std::move` existiert, um einen nicht rvalue-Ausdruck explizit in einen rvalue zu transformieren. Insbesondere wird der Ausdruck in einen x-Wert umgewandelt, denn selbst wenn es sich um einen identitätslosen pr-Wert-Ausdruck handelt, erhält er durch Übergabe als Parameter an `std::move` Identität (den Parameternamen der Funktion) und wird zum x-Wert.

Folgendes berücksichtigen:

```
std::string str("init");                      //1
std::string test1(str);                       //2
std::string test2(std::move(str));            //3

str = std::string("new value");               //4
```

```
std::string &&str_ref = std::move(str);        //5
std::string test3(str_ref);                    //6
```

`std::string` hat einen Konstruktor, der einen einzelnen Parameter vom Typ `std::string&&` , der üblicherweise als "Move-Konstruktor" bezeichnet wird. Die Wertkategorie des Ausdrucks `str` ist jedoch kein r-Wert (insbesondere ein l-Wert), daher kann diese Konstruktorüberladung nicht aufgerufen werden. Stattdessen ruft sie den `const std::string&` Konstruktor auf.

Zeile 3 verändert die Dinge. Der Rückgabewert von `std::move` ist ein `T&&` , wobei `T` der Basistyp des übergebenen Parameters ist. Daher gibt `std::move(str) std::string&&` . Ein Funktionsaufruf, dessen Rückgabewert ein rvalue-Verweis ist, ist ein rvalue-Ausdruck (insbesondere ein xvalue). Daher kann er den Bewegungskonstruktor von `std::string` aufrufen. Nach Zeile 3 wurde aus `str` verschoben (dessen Inhalt ist jetzt undefiniert).

Zeile 4 übergibt ein temporäres `std::string` an den Zuweisungsoperator von `std::string` . Dies hat eine Überladung, die eine `std::string&&` . Der Ausdruck `std::string("new value")` ist ein rvalue-Ausdruck (insbesondere ein prvalue). Daher kann er diese Überladung aufrufen. Daher wird das temporäre `str` in `str` verschoben, und der undefinierte Inhalt wird durch bestimmte Inhalte ersetzt.

Zeile 5 erstellt eine benannte rvalue-Referenz mit dem Namen `str_ref` , die auf `str` verweist. Hier werden Wertkategorien verwirrend.

Während `str_ref` ein rvalue-Verweis auf `std::string` , ist die Wertkategorie des Ausdrucks `str_ref` *kein rvalue* . Es ist ein lvalue Ausdruck. Ja wirklich. Aus diesem Grund kann der Bewegungskonstruktor von `std::string` mit dem Ausdruck `str_ref` . Zeile 6 *kopiert* daher den Wert von `str` in `test3` .

Um es zu verschieben, müssten wir erneut `std::move` .

Wertkategorien online lesen: https://riptutorial.com/de/cplusplus/topic/763/wertkategorien

Kapitel 145: Zeiger

Einführung

Ein Zeiger ist eine Adresse, die sich auf einen Speicherplatz bezieht. Sie werden häufig verwendet, um Funktionen oder Datenstrukturen zu ermöglichen, den Speicher zu kennen und zu modifizieren, ohne den betreffenden Speicher kopieren zu müssen. Zeiger können sowohl mit primitiven (integrierten) als auch benutzerdefinierten Typen verwendet werden.

Zeiger verwenden die Operatoren "dereference" `*` , "address of" `&` und "arrow" `->` . Die Operatoren '*' und '->' werden für den Zugriff auf den Speicher verwendet, auf den gezeigt wird, und der Operator `&` dient zum Abrufen einer Adresse im Speicher.

Syntax

- <Datentyp> * <Variablenname>;
- <Datentyp> * <Variablenname> = & <Variablenname des gleichen Datentyps>;
- <Datentyp> * <Variablenname> = <Wert desselben Datentyps>;
- int * foo; // Ein Zeiger, der auf einen ganzzahligen Wert zeigt
- int * bar = & myIntVar;
- langer * Stab [2];
- long * bar [] = {& myLongVar1, & myLongVar2}; // Entspricht: long * bar [2]

Bemerkungen

Beachten Sie Probleme beim Deklarieren mehrerer Zeiger in derselben Zeile.

```
int* a, b, c; //Only a is a pointer, the others are regular ints.

int* a, *b, *c; //These are three pointers!

int *foo[2]; //Both *foo[0] and *foo[1] are pointers.
```

Examples

Zeigergrundlagen

C ++ 11

Anmerkung: In allen folgenden `nullptr` wird das Vorhandensein der C ++ 11-Konstante `nullptr` angenommen. Ersetzen Sie in früheren Versionen `nullptr` durch `NULL` , die Konstante, die zuvor eine ähnliche Rolle gespielt hat.

Zeigervariable erstellen

Eine Zeigervariable kann mit der spezifischen `*` Syntax erstellt werden, z. B. `int *pointer_to_int;` .

Wenn eine Variable vom *Zeigertyp* (`int *`) ist, enthält sie nur eine Speicheradresse. Die Speicheradresse ist der Ort, an dem Daten des *zugrunde liegenden Typs* (`int`) gespeichert werden.

Der Unterschied ist deutlich, wenn Sie die Größe einer Variablen mit der Größe eines Zeigers mit demselben Typ vergleichen:

```
// Declare a struct type `big_struct` that contains
// three long long ints.
typedef struct {
    long long int foo1;
    long long int foo2;
    long long int foo3;
} big_struct;

// Create a variable `bar` of type `big_struct`
big_struct bar;
// Create a variable `p_bar` of type `pointer to big_struct`.
// Initialize it to `nullptr` (a null pointer).
big_struct *p_bar0 = nullptr;

// Print the size of `bar`
std::cout << "sizeof(bar) = " << sizeof(bar) << std::endl;
// Print the size of `p_bar`.
std::cout << "sizeof(p_bar0) = " << sizeof(p_bar0) << std::endl;

/* Produces:
    sizeof(bar) = 24
    sizeof(p_bar0) = 8
*/
```

Die Adresse einer anderen Variablen übernehmen

Zeiger können wie normale Variablen untereinander zugewiesen werden. In diesem Fall wird die **Speicheradresse** von einem Zeiger zu einem anderen kopiert, **nicht die tatsächlichen Daten** , auf die ein Zeiger zeigt.
Darüber hinaus können sie den Wert `nullptr` der einen `nullptr` darstellt. Ein Zeiger gleich `nullptr` enthält einen ungültigen Speicherplatz und bezieht sich daher nicht auf gültige Daten.

Sie können die Speicheradresse einer Variablen eines bestimmten Typs abrufen, indem Sie der Variablen die *Adresse des* Operators `&` voranstellen. Der von `&` Wert ist ein Zeiger auf den zugrunde liegenden Typ, der die Speicheradresse der Variablen enthält (die gültigen Daten ist **, solange die Variable nicht den Gültigkeitsbereich verlässt**).

```
// Copy `p_bar0` into `p_bar_1`.
big_struct *p_bar1 = p_bar0;

// Take the address of `bar` into `p_bar_2`
big_struct *p_bar2 = &bar;

// p_bar1 is now nullptr, p_bar2 is &bar.

p_bar0 = p_bar2;

// p_bar0 is now &bar.

p_bar2 = nullptr;

// p_bar0 == &bar
// p_bar1 == nullptr
// p_bar2 == nullptr
```

Im Gegensatz zu Referenzen:

- Durch das Zuweisen von zwei Zeigern wird der Speicher, auf den der zugewiesene Zeiger verweist, nicht überschrieben.
- Zeiger können Null sein.
- Die *Adresse des* Betreibers ist ausdrücklich erforderlich.

Auf den Inhalt eines Zeigers zugreifen

Wie nehmen Sie eine Adresse erfordert `&` sowie den Zugriff auf Inhalte erfordert die Verwendung des *Dereferenzierungsoperator* `*` , als Präfix. Wenn ein Zeiger dereferenziert wird, wird er zu einer Variablen des zugrunde liegenden Typs (eigentlich eine Referenz darauf). Es kann dann gelesen und geändert werden, wenn nicht `const` .

```
(*p_bar0).foo1 = 5;

// `p_bar0` points to `bar`. This prints 5.
std::cout << "bar.foo1 = " << bar.foo1 << std::endl;

// Assign the value pointed to by `p_bar0` to `baz`.
big_struct baz;
baz = *p_bar0;

// Now `baz` contains a copy of the data pointed to by `p_bar0`.
// Indeed, it contains a copy of `bar`.

// Prints 5 as well
std::cout << "baz.foo1 = " << baz.foo1 << std::endl;
```

Die Kombination aus `*` und dem Operator `.` wird mit `->` abgekürzt:

```
std::cout << "bar.foo1 = " << (*p_bar0).foo1 << std::endl; // Prints 5
std::cout << "bar.foo1 = " <<  p_bar0->foo1  << std::endl; // Prints 5
```

Ungültige Zeiger ableiten

Beim Dereferenzieren eines Zeigers sollten Sie sicherstellen, dass er auf gültige Daten zeigt. Das Dereferenzieren eines ungültigen Zeigers (oder eines Nullzeigers) kann zu einer Verletzung des Speicherzugriffs oder zum Lesen oder Schreiben von Speicherdaten führen.

```
big_struct *never_do_this() {
   // This is a local variable. Outside `never_do_this` it doesn't exist.
   big_struct retval;
   retval.foo1 = 11;
   // This returns the address of `retval`.
   return &retval;
   // `retval` is destroyed and any code using the value returned
   // by `never_do_this` has a pointer to a memory location that
   // contains garbage data (or is inaccessible).
}
```

In einem solchen Szenario geben `g++` und `clang++` die Warnungen korrekt aus:

```
(Clang) warning: address of stack memory associated with local variable 'retval' returned [-
Wreturn-stack-address]
(Gcc)   warning: address of local variable ‘retval’ returned [-Wreturn-local-addr]
```

Daher ist Vorsicht geboten, wenn Zeiger Argumente von Funktionen sind, da sie null sein könnten:

```
void naive_code(big_struct *ptr_big_struct) {
    // ... some code which doesn't check if `ptr_big_struct` is valid.
    ptr_big_struct->foo1 = 12;
}

// Segmentation fault.
naive_code(nullptr);
```

Zeigeroperationen

Es gibt zwei Operatoren für Zeiger: Address-of-Operator (&): Liefert die Speicheradresse seines Operanden. Operator für Inhalt (von) (*): Gibt den Wert der Variablen an der vom Operator angegebenen Adresse zurück.

```
int var = 20;
int *ptr;
ptr = &var;

cout << var << endl;
//Outputs 20 (The value of var)

cout << ptr << endl;
//Outputs 0x234f119 (var's memory location)

cout << *ptr << endl;
//Outputs 20(The value of the variable stored in the pointer ptr
```

Das Sternchen (*) wird bei der Deklaration eines Zeigers verwendet, um anzuzeigen, dass es sich um einen Zeiger handelt. Verwechseln Sie dies nicht mit dem **Dereferenzierungsoperator** , mit dem der Wert an der angegebenen Adresse abgerufen wird. Es sind einfach zwei verschiedene Dinge, die mit demselben Zeichen dargestellt werden.

Zeigerarithmetik

Inkrement / Dekrement

Ein Zeiger kann inkrementiert oder dekrementiert werden (Präfix und Postfix). Durch das Inkrementieren eines Zeigers wird der Zeigerwert an das Element im Array angehängt, und zwar ein Element nach dem aktuell angezeigten Element. Durch das Dekrementieren eines Zeigers wird er zum vorherigen Element im Array verschoben.

Die Zeigerarithmetik ist nicht zulässig, wenn der Typ, auf den der Zeiger zeigt, nicht vollständig ist. `void` ist immer ein unvollständiger Typ.

```
char* str = new char[10]; // str = 0x010
++str;                    // str = 0x011  in this case sizeof(char) = 1 byte

int* arr = new int[10];   // arr = 0x00100
++arr;                    // arr = 0x00104 if sizeof(int) = 4 bytes

void* ptr = (void*)new char[10];
++ptr;    // void is incomplete.
```

Wenn ein Zeiger auf das Endelement inkrementiert wird, zeigt der Zeiger auf ein Element nach dem Ende des Arrays. Ein solcher Zeiger kann nicht dereferenziert werden, er kann jedoch dekrementiert werden.

Durch das Inkrementieren eines Zeigers auf das einzeilige Element in dem Array oder das Dekrementieren eines Zeigers auf das erste Element in einem Array wird ein undefiniertes Verhalten erzielt.

Ein Zeiger auf ein Nicht-Array-Objekt kann für die Zeigerarithmetik behandelt werden, als wäre es ein Array der Größe 1.

Addition Subtraktion

Integer-Werte können Zeigern hinzugefügt werden. Sie fungieren als Inkrementieren, jedoch um eine bestimmte Zahl statt um 1. Ganzzahlige Werte können ebenfalls von Zeigern abgezogen werden, um als Zeigerdekrementierung zu fungieren. Wie beim Inkrementieren / Dekrementieren muss der Zeiger auf einen vollständigen Typ zeigen.

```
char* str = new char[10];  // str = 0x010
str += 2;                  // str = 0x010 + 2 * sizeof(char) = 0x012
```

```
int* arr = new int[10];    // arr = 0x100
arr += 2;                  // arr = 0x100 + 2 * sizeof(int) = 0x108, assuming sizeof(int) ==
4.
```

Zeigerdifferenzierung

Die Differenz zwischen zwei Zeigern auf denselben Typ kann berechnet werden. Die zwei Zeiger müssen sich in demselben Arrayobjekt befinden. ansonsten undefiniertes Verhalten.

Wenn zwei Zeiger `P` und `Q` in demselben Array vorhanden sind, ist `P - Q` , wenn `P` das `i` te Element in dem Array ist und `Q` das `j` te Element ist, dann `i - j` . Der Typ des Ergebnisses ist `std::ptrdiff_t` von `<cstddef>` .

```
char* start = new char[10];  // str = 0x010
char* test = &start[5];
std::ptrdiff_t diff = test - start; //Equal to 5.
std::ptrdiff_t diff = start - test; //Equal to -5; ptrdiff_t is signed.
```

Zeiger online lesen: https://riptutorial.com/de/cplusplus/topic/3056/zeiger

Kapitel 146: Zifferntrennzeichen

Examples

Zifferntrennzeichen

Numerische Literale mit mehr als ein paar Ziffern sind schwer zu lesen.

- 7237498123 aussprechen
- Vergleichen Sie 237498123 mit 237499123 für Gleichheit.
- Entscheiden Sie, ob 237499123 oder 20249472 größer ist.

`C++14` definiert Simple Quotation Mark `'` als Zifferntrennzeichen in Zahlen und benutzerdefinierten Literalen. Dies kann es für menschliche Leser einfacher machen, große Zahlen zu parsen.

C ++ 14

```
long long decn = 1'000'000'000ll;
long long hexn = 0xFFFF'FFFFll;
long long octn = 00'23'00ll;
long long binn = 0b1010'001111ll;
```

Anführungszeichen werden bei der Bestimmung des Werts ignoriert.

Beispiel:

- Die Literale `1048576` , `1'048'576` , `0X100000` , `0x10'0000` und `0'004'000'000` alle den gleichen Wert.
- Die Literale `1.602'176'565e-19` und `1.602176565e-19` haben den gleichen Wert.

Die Position der einfachen Anführungszeichen ist irrelevant. Alle folgenden sind gleichwertig:

C ++ 14

```
long long a1 = 12345678911;
long long a2 = 123'456'78911;
long long a3 = 12'34'56'78'911;
long long a4 = 12345'67891ll;
```

Es ist auch in `user-defined` Literalen zulässig:

C ++ 14

```
std::chrono::seconds tiempo = 1'674'456s + 5'300h;
```

Zifferntrennzeichen online lesen:
https://riptutorial.com/de/cplusplus/topic/10595/zifferntrennzeichen

Kapitel 147: Zufallszahlengenerierung

Bemerkungen

Die Erzeugung von `<random>` in C ++ wird vom Header `<random>` bereitgestellt. Dieser Header definiert Zufallsgeräte, Pseudozufallsgeneratoren und Verteilungen.

Zufällige Geräte geben vom Betriebssystem bereitgestellte Zufallszahlen zurück. Sie sollten entweder zur Initialisierung von Pseudozufallsgeneratoren oder direkt für kryptographische Zwecke verwendet werden.

Pseudozufallsgeneratoren geben basierend auf ihrem Anfangsstart ganzzahlige Pseudozufallszahlen zurück. Der Pseudozufallszahlenbereich umfasst typischerweise alle Werte eines vorzeichenlosen Typs. Alle Pseudo-Zufallsgeneratoren in der Standardbibliothek geben für alle Plattformen die gleichen Zahlen für denselben Anfangswert aus.

Verteilungen verbrauchen Zufallszahlen von Pseudozufallsgeneratoren oder Zufallsvorrichtungen und erzeugen Zufallszahlen mit der erforderlichen Verteilung. Distributionen sind nicht plattformunabhängig und können unterschiedliche Nummern für dieselben Generatoren mit demselben Start-Seed auf verschiedenen Plattformen erzeugen.

Examples

Echter Zufallswertgenerator

Um echte Zufallswerte zu erzeugen, die für die Kryptographie verwendet werden können, muss `std::random_device` als Generator verwendet werden.

```
#include <iostream>
#include <random>

int main()
{
   std::random_device crypto_random_generator;
   std::uniform_int_distribution<int> int_distribution(0,9);

   int actual_distribution[10] = {0,0,0,0,0,0,0,0,0,0};

   for(int i = 0; i < 10000; i++) {
       int result = int_distribution(crypto_random_generator);
       actual_distribution[result]++;
   }

   for(int i = 0; i < 10; i++) {
       std::cout << actual_distribution[i] << " ";
   }

   return 0;
}
```

`std::random_device` wird auf dieselbe Weise verwendet wie ein Pseudo-Zufallswertgenerator.

`std::random_device` **kann jedoch im Hinblick auf eine implementierungsdefinierte Pseudozufallszahlen-Engine implementiert werden,** wenn eine nicht deterministische Quelle (z. B. ein Hardwaregerät) für die Implementierung nicht verfügbar ist.

Das Erkennen solcher Implementierungen sollte über die `entropy` Member-Funktion möglich sein (die Null zurückgibt, wenn der Generator vollständig deterministisch ist), aber viele gängige Bibliotheken (sowohl GCCs libstdc ++ als auch LLVMs libc ++) geben immer null zurück, selbst wenn sie qualitativ hochwertige externe Zufälligkeiten verwenden .

Generierung einer Pseudo-Zufallszahl

Ein Pseudo-Zufallszahlengenerator generiert Werte, die basierend auf zuvor generierten Werten geschätzt werden können. Mit anderen Worten: es ist deterministisch. Verwenden Sie keinen Pseudo-Zufallszahlengenerator in Situationen, in denen eine echte Zufallszahl erforderlich ist.

```
#include <iostream>
#include <random>

int main()
{
   std::default_random_engine pseudo_random_generator;
   std::uniform_int_distribution<int> int_distribution(0, 9);

   int actual_distribution[10] = {0,0,0,0,0,0,0,0,0,0};

   for(int i = 0; i < 10000; i++) {
       int result = int_distribution(pseudo_random_generator);
       actual_distribution[result]++;
   }

   for(int i = 0; i <= 9; i++) {
       std::cout << actual_distribution[i] << " ";
   }

   return 0;
}
```

Dieser Code erstellt einen Zufallszahlengenerator und eine Verteilung, die mit gleicher Wahrscheinlichkeit Ganzzahlen im Bereich [0,9] generiert. Es zählt dann, wie oft jedes Ergebnis generiert wurde.

Der Vorlagenparameter von `std::uniform_int_distribution<T>` gibt den Typ der Ganzzahl an, die generiert werden soll. Verwenden Sie `std::uniform_real_distribution<T>` , um Floats oder Doubles zu generieren.

Verwendung des Generators für mehrere Distributionen

Der Zufallszahlengenerator kann (und sollte) für mehrere Verteilungen verwendet werden.

```
#include <iostream>
```

```
#include <random>

int main()
{
   std::default_random_engine pseudo_random_generator;
   std::uniform_int_distribution<int> int_distribution(0, 9);
   std::uniform_real_distribution<float> float_distribution(0.0, 1.0);
   std::discrete_distribution<int> rigged_dice({1,1,1,1,1,100});

   std::cout << int_distribution(pseudo_random_generator) << std::endl;
   std::cout << float_distribution(pseudo_random_generator) << std::endl;
   std::cout << (rigged_dice(pseudo_random_generator) + 1) << std::endl;

   return 0;
}
```

In diesem Beispiel ist nur ein Generator definiert. Anschließend wird ein Zufallswert in drei verschiedenen Verteilungen generiert. Die `rigged_dice` Verteilung einen Wert zwischen 0 und 5, erzeugt aber fast erzeugt immer ein `5` , weil die Chance ein generieren `5` ist `100 / 105` .

Zufallszahlengenerierung online lesen:
https://riptutorial.com/de/cplusplus/topic/1541/zufallszahlengenerierung

Credits

S. No	Kapitel	Contributors
1	Erste Schritte mit C ++	Adhokshaj Mishra, ankit dassor, aquirdturtle, ArchbishopOfBanterbury, Bakhtiar Hasan, Bart van Nierop, Ben H, Bo Persson, Brandon, Brian , BullshitPingu, cb4, celtschk, Cheers and hth. - Alf, chrisb2244, Cody Gray, Community, cpatricio, Curious, Daemon, Daksh Gupta, Danh, darkpsychic, David Bippes, David G., DeepCoder, Dim_ov, dlemstra, Donald Duck, Dr t, Dylan Little, Edward, emlai, Erick Q., ethanwu10, Fantastic Mr Fox, Florian, GIRISH kuniyal, greatwolf, honk, Humam Helfawi, Hurkyl, Ilyas Mimouni, Isak Combrinck, itzmukeshy7, Jason Watkins, JedaiCoder, Jerry Coffin, Jim Clark, Johan Lundberg, Jon Harper, jotik, Justin, Justin Time, JVApen, K48, Ken Y-N, Keshav Sharma, kiner_shah, krOoze, Leandros, maccard, Malcolm, Malick, Manan Sharma, manetsus, manlio, Marco A., Mark Gardner, MasterHD, Matt, Matt Lord, mnoronha , Muhammad Aladdin, Mustaghees, muXXmit2X, mynameisausten, Nathan Osman, Neil A., Nemanja Boric, neuro, Nicol Bolas, no ɥʎdʎzɐƆ, Optimus Prime, Pavel Strakhov, Peter, Praetorian, Qchmqs, Quirk, RamenChef, Rushikesh Deshpande, SajithP, Sam Cristall, Serikov, Shoe, SirGuy, Soapy, Soul_man, theo2003, ɥolɐəz əɥɬ qoq, Tom K, TriskalJM, Trizzle, UncleZeiv, VermillionAzure, Walter, Wen Qin, Wexiwa, πάντα ῥεῖ, パスカル
2	Ablaufsteuerung	anotherGatsby, Brian, JVApen, mkluwe, Qchmqs, RamenChef, Tejendra
3	Argumentabhängige Namenssuche	Fanael, Johannes Schaub - litb
4	Arithmitische Metaprogrammierung	Meena Alfons
5	Arrays	Cheers and hth. - Alf, Isak Combrinck, manlio, Matthew Brien, Wen Qin, Wolf, ΦXocę Π epeúpa ツ

6	Art Abzug	Barry, Brian, Emmanuel Mathi-Amorim
7	Atomtypen	JVApen, Stephen
8	Attribute	ibrahim5253, JVApen, Kerrek SB, MathSquared, SingerOfTheFall
9	Aufzählung	Denkkar, Fantastic Mr Fox, Jarod42, Nicol Bolas, SajithP, stackptr, T.C.
10	Ausdrücke falten	AndyG, Barry, cpplearner, Firas Moalla, Marco A., Rakete1111, T.C., Yakk
11	Ausdrucksvorlagen	Ajay, BigONotation, celtschk, defube, Jarod42, Jonathan Lee, Roland, T.C., Yakk
12	Ausnahmen	Alexey Guseynov, Brian, callyalater, Dr t, Jahid, Jarod42, Johan Lundberg, jotik, Martin Ba, Nemanja Boric, Null, Peter, Rakete1111, Ronen Ness
13	Ausrichtung	Brian, Marco A., Nicol Bolas
14	Auto	Artalus, Barry, celtschk, Daniele Pallastrelli, Edward, Igor Oks, Jarod42, Johan Lundberg, manlio, Yakk
15	Bauen Sie Systeme auf	Ami Tavory, celtschk, Florian, Jahid, Jason Watkins, Justin, JVApen, Nathan Osman, RamenChef, VermillionAzure
16	Beispiele für Client-Server	Abhinav Gauniyal
17	Benutzerdefinierte Literale	Brian, Cid1025, Jarod42, Roland, sigalor, sth
18	Bereiche	deepmax, Error - Syntactical Remorse
19	Bitfelder	Ajay, Perette Barella
20	Bit-Manipulation	A. Sarid, Barry, Cody Gray, CroCo, FedeWar, Jarod42, JVApen, manlio, tambre, Tarod, Trevor Hickey, Алексей Неудачин
21	Bitoperatoren	Loki Astari, Mads Marquart, manlio, txtechhelp, Алексей Неудачин
22	C ++ - Container	John DiFini
23	C ++ - Funktion "Aufruf durch Wert" vs. "Aufruf durch Referenz"	Error - Syntactical Remorse, Henkersmann

24	C ++ 11-Speichermodell	NonNumeric
25	C ++ Streams	Ami Tavory, didiz, JVApen, mpromonet, Sergey
26	C Inkompatibilitäten	パスカル
27	Callable Objects	JVApen, turoni
28	Const Korrektheit	amanuel2, Justin Time
29	constexpr	Ajay, Brian, diegodfrf, mtb, Null
30	Datei I / O	anderas, ankit dassor, Anonymous1847, AProgrammer, Bakhtiar Hasan, bitek, Chachmu , ComicSansMS, didiz, Dietmar Kühl, Dr t, Emanuel Vintilă, Galik, honk, Hurkyl, Jérémie Bolduc, John Strood, JVApen, Loki Astari, manlio, Mathieu K., MikeMB, mindriot, Nicol Bolas, nwp, patmanpato, Rakete1111, RomCoo, Serikov, sheng09, shrike, svgspnr, Алексей Неудачин
31	Datenstrukturen in C ++	Gaurav Sehgal
32	Datum und Uhrzeit mit Header	Edward, marcinj, Naor Hadar, RamenChef
33	decltype	Ajay
34	Deklaration verwenden	Brian
35	Der ISO-C ++ - Standard	Bakhtiar Hasan, Barry, C.W.Holeman II, ComicSansMS, didiz, diegodfrf, Guillaume Pascal, Ivan Kush, Johan Lundberg, Justin Time, JVApen, manlio, Marco A., MSalters, Nicol Bolas, sth, vishal
36	Der This Pointer	amanuel2, Justin Time, RamenChef
37	Designmuster-Implementierung in C ++	Antonio Barreto, datosh, didiz, Jarod42, JVApen, Nikola Vasilev
38	Die Regel von Drei, Fünf und Null	Adrien Descamps, Barry, ChrisN, hello, honk, Johan Lundberg, Justin Time, JVApen, Loki Astari, mpromonet, Nicol Bolas, Nirmal4G, NonNumeric, Null, Peter, relgukxilef, Scott Weldon, T.C., TriskalJM, Venemo
39	Eigenschaften eingeben	Jarod42, silvergasp, TartanLlama
40	Eine Definitionsregel (ODR)	Brian, Jarod42

41	Einfädeln	Alejandro, amchacon, Brian, CaffeineToCode, ComicSansMS, Dair, defube, didiz, Diligent Key Presser, Galik, James Adkison, james large, Jason Watkins, Jeremi Podlasek, mpromonet, Niall, nwp, Rakete1111, Stephen Cross, Sumurai8, Yakk, ysdx, Yuushi
42	Elision kopieren	Nicol Bolas, TartanLlama
43	Explizite Typkonvertierungen	4444, Brian, JVApen, Nikola Vasilev
44	Fließkomma-Arithmetik	Xirema
45	Freund-Schlüsselwort	Perette Barella, Sergey
46	Funktionsüberladung	Bakhtiar Hasan, Barry, Cody Gray, CoffeeandCode, didiz, Galik, Jatin, Johannes Schaub - litb, Justin Time, JVApen, Rakete1111, Sean, Sumurai8, Tim Straubinger
47	Funktionsvorlage überladen	Johannes Schaub - litb, Kunal Tyagi, RamenChef
48	Futures und Versprechen	didiz, Nicol Bolas
49	Geben Sie Schlüsselwörter ein	Brian, Justin Time, Omnifarious, RamenChef
50	Gewerkschaften	manlio, ThyReaper, txtechhelp
51	Grundlegende Eingabe / Ausgabe in C ++	Daemon, Nicol Bolas, Владимир Стрелец
52	Grundtyp-Schlüsselwörter	amanuel2, Brian, Kerrek SB, RamenChef
53	Häufige Compile / Linker-Fehler (GCC)	Asu, immerhart
54	Header-Dateien	RamenChef, VermillionAzure
55	Hinterlegter Rückgabetyp	Brian, define cindy const, Torbjörn
56	Hinweise auf Mitglieder	John Burger, start2learn
57	Implementierungsdefiniertes Verhalten	2501, Bo Persson, Brian, Dutow, Jahid, Jarod42, jotik, Justin Time, lz96, manlio, Nicol Bolas, Peter
58	Inline-Funktionen	amanuel2, Aravind .KEN, Bim, Brian, legends2k
59	Inline-Variablen	Brian

60	Intelligente Zeiger	Abyx, Ajay, Alexey Voytenko, anderas, Barry, CaffeineToCode, Christopher Oezbek, Cody Gray, ComicSansMS, cpplearner, Daksh Gupta , Danh, Daniele Pallastrelli, DeepCoder, Edward, emlai, foxcub, Francis Cugler, honk, Jack Zhou, Jared Payne, Jarod42, Johan Lundberg, Johannes Schaub - litb, jotik, Justin, JVApen, Kerrek SB, King's jester, Loki Astari, manlio, Marco A., MC93, Menasheh, Meysam, PcAF, Rakete1111, Reuben Thomas, Richard Dally, rodrigo, Roland, sami1592, sth, Sumurai8, tysonite, user3684240, Xirema, Yakk
61	Internationalisierung in C ++	John Bargman
62	Iteration	Brian, Daniel Käfer, Emmanuel Mathi-Amorim, marquesm91, RamenChef
63	Iteratoren	Barry, chrisb2244, cute_ptr, Daniel Jour, Edgar Rokyan, EvgeniyZh, fbrereto, Gal Dreiman, Gaurav Kumar Garg, GIRISH kuniyal, honk, Hurkyl, JPNotADragon, JVApen, Mike H-R, Null , Oz., Sergey, Serikov, tilz0R, Yakk
64	Klassen / Strukturen	Alexey Voytenko, anderas, aquirdturtle, Brian, callyalater, chrisb2244, Colin Basnett, Dan Hulme, darkpsychic, Dragma, Fantastic Mr Fox, Firas Moalla, Jarod42, Jerry Coffin, jotik, Justin Time, Kerrek SB, Nicol Bolas, Null, OliPro007, PcAF, Ph03n1x, pingul, Rakete1111, Sándor Mátyás Márton, Sergey, silvergasp, Skywrath, Yakk
65	Kompilieren und Bauen	4444, Adhokshaj Mishra, Ami Tavory, ArchbishopOfBanterbury, Barry, Ben Steffan, celtschk, Curious, Donald Duck, Dr t, elvis.dukaj, Fantastic Mr Fox, Florian, greatwolf , Griffin, Isak Combrinck, Jahid, Jarod42, Jason Watkins, Johan Lundberg, jotik, Justin, Justin Time, JVApen, madduci, Malick, manetsus, manlio, Matt, Michael Gaskill, Morten Kristensen, MSD, muXXmit2X, n.m., Nathan Osman, Nemanja Boric, Peter, Quirk, Richard Dally, Sergey, Tharindu Kumara, Toby, Trygve Laugstøl, VermillionAzure
66	Komponententest in C ++	elvis.dukaj, VermillionAzure

67	Konstante Klassenmitgliederfunktionen	Vijayabhaskarreddy CH, Yakk
68	Kopieren vs Zuordnung	amanuel2, Roland
69	Kopplungsspezifikationen	Brian
70	Lambdas	Adi Lester, Aganju, Ajay, alain, anderas, Andrea Corbelli, Barry, bcmpinc, Brian, Christopher Oezbek, Community, cpplearner, derekerdmann, Edd, Falias, Firas Moalla, honk, Jean-Baptiste Yunès, Johan Lundberg, Johannes Schaub - litb, John Slegers, JVApen, Loki Astari, Loufylouf, M. Viaz, Mike Dvorkin, Nicol Bolas, Patryk, Praetorian, Rakete1111, RamenChef, Ryan Haining, Sergio, Serikov, Snowhawk, teivaz, Yakk, ygram
71	Layout der Objekttypen	Brian, Justin Time
72	Literale	Brian, Nikola Vasilev, RamenChef
73	Mehrere Werte aus einer Funktion zurückgeben	aaronsnoswell, Bakhtiar Hasan, Barry, bitek, celtschk, Christopher Oezbek, Community, DeepCoder, Dr t, Ela782, Fantastic Mr Fox, Galik, honk, J_T, Jarod42, Johan Lundberg, Johannes Schaub - litb, John Slegers, Jon Chesterfield, Kevin Katzke, Let_Me_Be, Loki Astari, M. Sadeq H. E., manetsus, Menasheh, Michael Gaskill, mnoronha, Niall, Nicol Bolas, Null, Peter, Rakete1111, Richard Forrest, Ryan Hilbert, Stephen, T.C., templatetypedef, tenpercent, user3384414, Yakk, Ze Rubeus, パスカル
74	Metaprogrammierung	anderas, Barry, Brian, celtschk, Colin Basnett, DawidPi, deepmax, dmi_, Holt, Jarod42, Justin, manlio, Matthieu M., Nicol Bolas, Oz., rhynodegreat, rtmh, sth, TartanLlama, Venki, W.F., ysdx, πάντα ῥεῖ
75	Mutexe	didiz, hyoslee, JVApen
76	Namensräume	anderas, Andrea Chua, Barry, Brian, DeepCoder, emlai, Isak Combrinck, Jarod42, Jérémy Roy, Johannes Schaub - litb, Julien-L, JVApen, Nicol Bolas, Null, Rakete1111, randag, Roland, T.C., tenpercent, Yakk

77	Neugierig wiederkehrendes Vorlagenmuster (CRTP)	Barry, Brian, Gabriel, honk, Nicol Bolas, Ryan Haining
78	Nicht statische Memberfunktionen	Justin Time, RamenChef
79	Operator Vorrang	an0o0nym, Brian, didiz, JVApen, start2learn, turoni
80	Optimierung	chema989, ralismark
81	Optimierung in C ++	4444, JVApen, lorro, mindriot
82	Parallele Vergleiche von klassischen C ++ - Beispielen, die über C ++ vs C ++ 11 vs C ++ 14 vs C ++ 17 gelöst wurden	wasthishelpful
83	Parallelität mit OpenMP	Andrea Chua, JVApen, Nicol Bolas, Sumurai8
84	Parameterpakete	Marco A.
85	Perfekte Weiterleitung	In silico, Johannes Schaub - litb, Nicol Bolas, Roland
86	Pimpl-Idiom	Danh, Daniele Pallastrelli, emlai, Jordan Chapman, JVApen, manlio, Stephen Cross, Yakk
87	Polymorphismus	A. Sarid, Christophe, Jarod42, Jeremi Podlasek , Justin Time, manlio
88	Präprozessor	alain, callyalater, Cheers and hth. - Alf, CygnusX1, Fantastic Mr Fox, Fox, Francisco P. , Ian Ringrose, immerhart, InitializeSahib, Johan Lundberg, Justin, Justin Time, Ken Y-N, Kieran Chandler, krOoze, manlio, Marco A., Maxito, n.m., Nicol Bolas, Peter, phandinhlan, Richard Dally, Sean, signal, silvergasp, Sumurai8, T.C., Tanjim Hossain, tenpercent, The Philomath, Владимир Стрелец, パスカル
89	Profilierung	Ami Tavory, paul-g
90	RAII: Ressourcenakquisition ist Initialisierung	Barry, defube, Jarod42, JVApen, Loki Astari, Niall, Nicol Bolas, RamenChef, Sumurai8, Tannin
91	Refactoring-Techniken	asantacreu, Cody Gray, Edward, Jarod42, JVApen, RamenChef

92	Reguläre Ausdrücke	honk, Jonathan Mee, Justin, JVApen
93	Rekursion in C ++	celtschk, R_Kapp
94	Rekursiver Mutex	didiz
95	Resourcenmanagement	Anonymous1847
96	RTTI: Informationen zum Laufzeit-Typ	Brian, deepmax, Pankaj Kumar Boora, Roland, Savas Mikail KAPLAN
97	Rückgabetyp Kovarianz	Cheers and hth. - Alf, sorosh_sabz
98	Schleifen	ankit dassor, anotherGatsby, Barry, ChemiCalChems, Chris, ChrisN, Christian Rau, ColleenV, Debanjan Dhar, DrZoo, Edward, emlai, holmicz, honk, Johannes Schaub - litb, Justin Time, L.V.Rao, manlio, Nicholas, Nicol Bolas, Null, Ped7g, pmelanson, Pyves, Rakete1111, Sergey, sp2danny, user1336087, VladimirS, Yakk
99	Schlüsselwort const	Barry, Jarod42, Jatin, Justin, Podgorskiy, tenpercent, ThyReaper
100	Schlüsselwörter	ADITYA, amanuel2, Brian, Danh, John London, Justin Time, JVApen, Kerrek SB, Loki Astari, manlio, Marco A., Nicol Bolas, OliPro007, Rakete1111, RamenChef, Roland, start2learn
101	Semantik verschieben	Barry, Cheers and hth. - Alf, ChemiCalChems, David Doria, didiz, Guillaume Racicot, Justin Time, JVApen
102	Semaphor	didiz
103	SFINAE (Substitutionsfehler ist kein Fehler)	Barry, Fox, Jarod42, Jason R, Jonathan Lee, Luc Danton, sp2danny, SU3, w1th0utnam3, Xosdy, Yakk
104	Singleton Design Pattern	deepmax, Galik, Jarod42, Johan Lundberg, JVApen, Stradigos
105	Sortierung	anatolyg, Barry, Daniel, Ivan Kush, maccard, manetsus, manlio, MikeMB, MKAROL, Nicol Bolas, Patrick, Ravi Chandra, SajithP, timrau, Trevor Hickey
106	Speicherklassenspezifizierer	Brian, start2learn

107	Speicherverwaltung	Andrei, Brian, callyalater, Daksh Gupta, Galik, JVApen, madduci, nnrales, RamenChef, ThyReaper
108	Spezielle Mitgliederfunktionen	Barry, krOoze, OliPro007, Reuben Thomas, TriskalJM
109	Standard-Bibliotheksalgorithmen	Ami Tavory, Barry, Daniel, Duly Kinsky, Edgar Rokyan, Guillaume Pascal, Jarod42, NinjaDeveloper, Patryk Obara, Peter, Riom
110	static_assert	Jarod42, JVApen, lorro, Marco A., Richard Dally, T.C.
111	std :: any	demonplus, Marco A.
112	std :: array	CinCout, Daksh Gupta, Dinesh Khandelwal, Error - Syntactical Remorse, Malcolm, Nikola Vasilev, plasmacel
113	std :: atomics	Nikola Vasilev
114	std :: forward_list	Nikola Vasilev
115	std :: function: Um ein Element aufzurufen, das aufrufbar ist	elimad, Evgeniy, Nicol Bolas, Tarod
116	std :: integer_sequence	Dietmar Kühl
117	std :: iomanip	kiner_shah, Nikola Vasilev, Yakk
118	std :: map	Andrea Corbelli, ankit dassor, ChrisN, CinCout, ComicSansMS, CygnusX1, davidsheldon, diegodfrf, Fantastic Mr Fox, foxcub, Galik, honk , jmmut, manetsus, manlio, Meysam, Naveen Mittal, Null, Peter, Richard Dally, rick112358, Savan Morya, user1336087, vdaras, VolkA, Wyzard
119	std :: optional	Barry, diegodfrf, Jahid, Jared Payne, JVApen, Null, Yakk
120	std :: pair	Ajay, Bim, demonplus, kiner_shah, Nikola Vasilev, Ravi Chandra
121	std :: set und std :: multiset	G-Man, JVApen, Mikitori
122	std :: string	1337ninja, 3442, Andrea Corbelli, Barry, Bim, caps, Christopher Oezbek, cpplearner, crea7or, Curious, drov, Edward, Emil Rowland, emlai,

		Fantastic Mr Fox, fbrereto, ggrr, Holt, honk, immerhart, Jack, Jahid, Jerry Coffin, Jonathan Mee, jotik, JPNotADragon, jpo38, Justin, JVApen, Ken Y-N, Leandros, Loki Astari, manetsus, manlio, Marc.2377, Matthew, Matthieu M., Meysam, Michael Gaskill, mpromonet, Niall, Null, Rakete1111, RamenChef, Richard Dally, SajithP, Serikov, sigalor, Skipper, Soapy, sth, T.C., Tharindu Kumara, Trevor Hickey, user1336087, user2176127, W.F., Wolf, Yakk
123	std :: variant	Yakk
124	std :: vector	2power10, A. Sarid, Aaron Stein, alain, Alex Logan, Ami Tavory, anatolyg, anderas, Andy, AndyG, arunmoezhi, Bakhtiar Hasan, Barry, Benjamin Lindley, bluefog, bone, CHess, CinCout, Cody Gray, Colin Basnett, ComicSansMS, Community, cute_ptr, Daksh Gupta, Daniel, Daniel Stradowski, Dario, David G., David Yaw, DeepCoder, diegodfrf, dkg, Dr t, Duly Kinsky, Ed Cottrell, Edward, ehudt, emlai, enrico.bacis, Falias, Fantastic Mr Fox, Fox, foxcub, gaazkam, Galik, gartenriese, granmirupa, Holt, honk, Hurkyl, iliketocode, immerhart, Isak Combrinck, Jarod42, Jason Watkins, JHBonarius, Johan Lundberg, John Slegers, jotik, jpo38, JVApen, Kevin Katzke, krOoze, Loki Astari, lordjohncena, manetsus, manlio, Marco A., Matt, Michael Gaskill, Misha Brukman, MotKohn, Motti, mtk, NageN, Niall, Nicol Bolas, Null, patmanpato, Paul Beckingham, paul-g, Ped7g, Praetorian, Pyves, R. Martinho Fernandes, Rakete1111, Randy Taylor, Richard Dally, Roddy, Romain Vincent, Rushikesh Deshpande, Ryan Hilbert, Saint-Martin, Samar Yadav, Samer Tufail, Sayakiss, Serikov, Shoe, silvergasp, Skipper, solidcell, Stephen, sth, strangeqargo, T.C., Tamarous, theo2003, Tom, towi, Trevor Hickey, TriskalJM, user1336087, user2176127, Vladimir Gamalyan, Wolf, Yakk
125	Stream-Manipulatoren	Nicol Bolas, Владимир Стрелец
126	Thread-Synchronisationsstrukturen	didiz, Galik, JVApen

127	Tools und Techniken zum Debuggen und Debuggen von C ++	Adam Trhon, JVApen, King's jester, Misgevolution
128	Typ Inferenz	Andrea Chua, Jim Clark
129	Typ löschen	Brian, celtschk, greatwolf, Jarod42, Yakk
130	Typedef- und Typ-Aliase	Brian
131	Überladung des Bedieners	ArchbishopOfBanterbury, Archie Gertsman, Ates Goral, Barry, Brian, Candlemancer, chrisb2244, defube, enzom83, James Adkison, Rakete1111, Sergey, start2learn, Xeverous, Yakk
132	Überlastauflösung	Barry, Brian, didiz, Johannes Schaub - litb
133	Unbekanntes Verhalten	AndreiM, Brian, Jarod42, Yakk
134	Unbenannte Typen	jotik, Roland, ThyReaper
135	Undefiniertes Verhalten	Ami Tavory, AndreiM, Ben Steffan, Brian, Cody Gray, cshu, Dovahkiin, Elias Kosunen, emlai, Emma X, FedeWar, fefe, ggrr, GIRISH kuniyal, Hiura, Jeremi Podlasek, Johannes Schaub - litb , JVApen, kd1508, Ken Y-N, manetsus, manlio, Marco A., Mat, mceo, Motti, Naor Hadar, nbro, Nicol Bolas, Peter, Rakete1111, ralismark, RamenChef, Sebastian Ärleryd, Tannin, Trevor Hickey, Tyler Durden
136	Variablendeklarationsschlüsselwörter	Brian, RamenChef, start2learn
137	veränderbares Schlüsselwort	Barry, Community, Dean Seo, start2learn, T.C., tenpercent
138	Verweise	Andrea Corbelli, Asu, Daksh Gupta, darkpsychic, rockoder
139	Verwenden von std :: unordered_map	tulak.hord
140	Virtuelle Elementfunktionen	0x5f3759df, Daksh Gupta, Johan Lundberg, Justin Time, Motti, Sergey, T.C.
141	Vorlagen	Barry, Benjy Kessler, Brian, callyalater, cb4, celtschk, CodeMouse92, Colin Basnett, DeepCoder, Diligent Key Presser, Eldritch Cheese, eXPerience, FedeWar, Gabriel, Greg, Holt, honk, J_T, Johannes Schaub - litb, Justin,

		JVApen, Loki Astari, M. Viaz, manlio, Maxito, MSalters, Nicol Bolas, Pontus Gagge, Praetorian, Rakete1111, Ricardo Amores, Ryan Haining, Sergey, SirGuy, Smeeheey, Sumurai8, user1887915, W.F., WMios, Wolf, πάντα ῥεῖ
142	Weitere undefinierte Verhalten in C ++	didiz
143	Wert und Referenzsemantik	JVApen, Nicol Bolas
144	Wertkategorien	Barry, ChemiCalChems, Curious, fefe, Johannes Schaub - litb, mnoronha, Nicol Bolas, Praetorian, SirGuy
145	Zeiger	Baron, daB0bby, FedeWar, Hindrik Stegenga, Nicol Bolas, Nitinkumar Ambekar, Pietro Saccardi, Reverie Wisp, West
146	Zifferntrennzeichen	diegodfrf, JVApen
147	Zufallszahlengenerierung	Ha., manlio, merlinND, Sumurai8